tek.com/keithley

Model 4200A-SCS KULT and KULT Extension

Programming

4200A-KULT-907-01 Rev. D May 2024

4200A-KULT-907-01D

KEITHLEY
A Tektronix Company

Model 4200A-SCS

KULT and KULT Extension Programming

Document number: 4200A-KULT-907-01 Rev. D May 2024

KEITHLEY
A Tektronix Company

Safety precautions

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

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

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

The types of product users are:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The ⊥ symbol indicates a connection terminal to the equipment frame.

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

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

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

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

Instrumentation and accessories shall not be connected to humans.

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

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

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

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

Safety precaution revision as of June 2018.

Table of contents

Section 1

Introduction

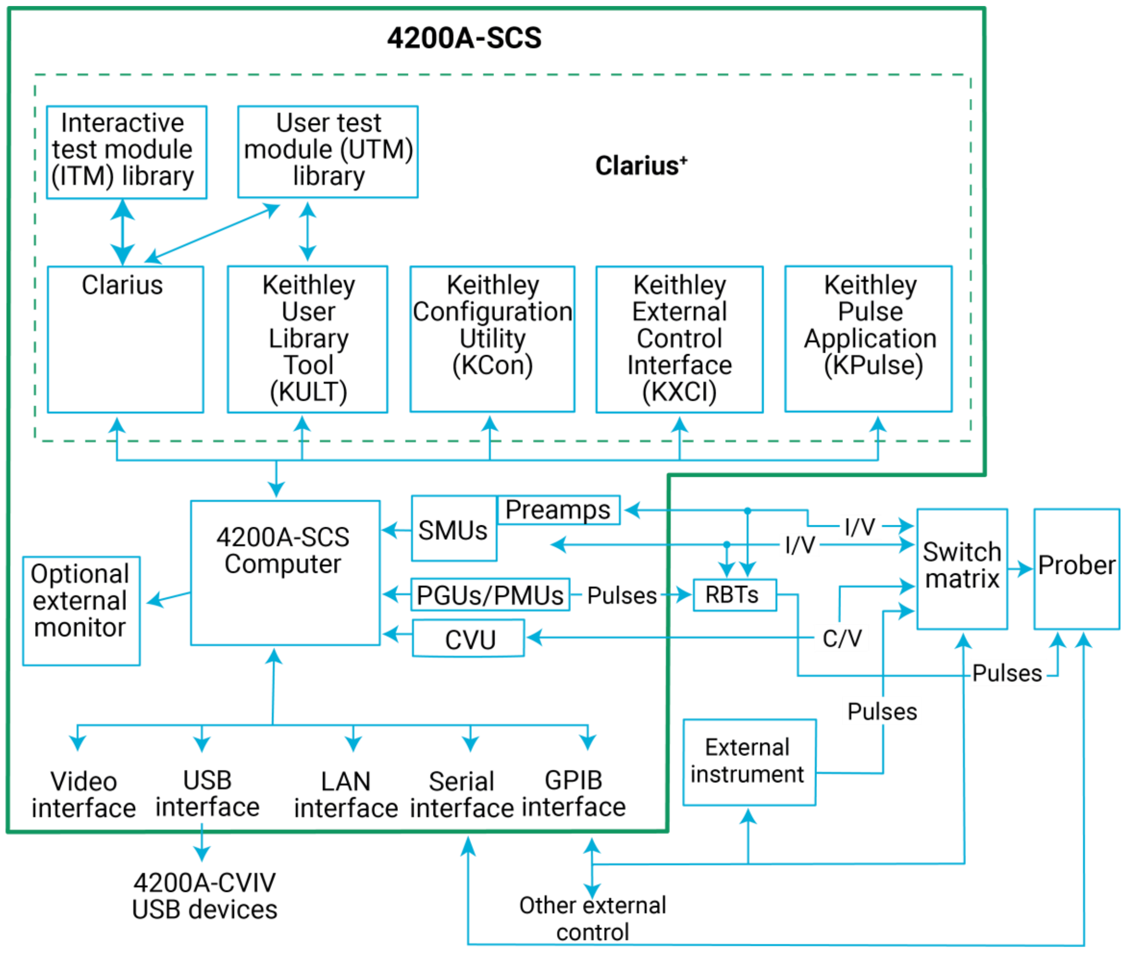
In this section:

Introduction

The Keithley User Library Tool (KULT) and the KULT Extension for Visual Studio Code are a few of the software tools provided with the Keithley Instruments Model 4200A-SCS. The 4200A-SCS is a customizable and fully integrated parameter analyzer that provides synchronized insight into current-voltage (I-V), capacitance-voltage (C-V), and ultra-fast pulsed I-V characterization. Its advanced digital sweep parameter analyzer combines speed and accuracy for deep sub-micron characterization.

The primary 4200A-SCS components and typical supported external components are illustrated in the following figure.

Figure 1: 4200A-SCS summary



KULT description

You can use the Keithley User Library Tool (KULT) and the KULT Extension for Visual Studio Code to create and manage user libraries. A user library is a collection of user modules. User modules are C programming language subroutines, also called functions. User libraries are created to control instrumentation, analyze data, or perform any other system automation task programmatically. Once a user library has been successfully built using KULT, its user modules can be executed using the Clarius software tool.

KULT provides a simple user interface that helps you effectively enter code, build a user module, and build a user library. KULT also provides management features for the user library, including menu commands to copy modules, copy libraries, delete modules, and delete library menu commands. KULT manages user libraries in a structured manner. You can create your own user libraries to extend the capabilities of the 4200A-SCS.

The KULT Extension for Visual Studio Code gives you the ability to write, compile, and debug user libraries outside of KULT. Combining the user-friendly Visual Studio Code editor with KULT creates an integrated development environment (IDE).

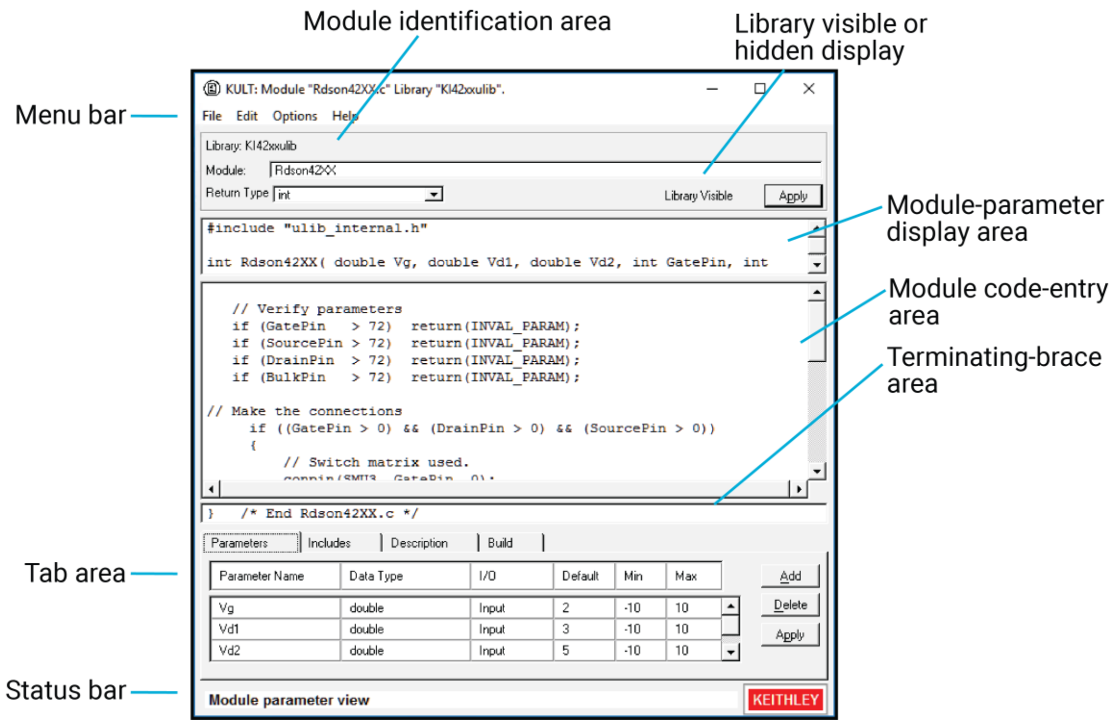
To execute a KULT user module in Clarius, you create a Clarius user test module (UTM) and connect it to the user module. Once this user module is connected to the UTM, the following occurs each time Clarius executes the UTM:

- Clarius dynamically loads the user module and the appropriate user library directory (`usrlib`).
- Clarius passes the user-module parameters (stored in the UTM) to the user module.
- Data generated by the user module is returned to the UTM for interactive analysis.

KULT interface description

The KULT interface is shown in the following figure. It provides all the menus, controls, and user-entry areas that you need to create, edit, view, and build a user library and to create, edit, view, and build a user module.

Figure 2: KULT interface overview



```
#include "ulib_internal.h"

int Rdson42XX( double Vg, double Vd1, double Vd2, int GatePin, int
```

```
   // Verify parameters
   if (GatePin   > 72)  return(INVAL_PARAM);
   if (SourcePin > 72)  return(INVAL_PARAM);
   if (DrainPin  > 72)  return(INVAL_PARAM);
   if (BulkPin   > 72)  return(INVAL_PARAM);

// Make the connections
     if ((GatePin > 0) && (DrainPin > 0) && (SourcePin > 0))
     {
         // Switch matrix used.
```

```
}   /* End Rdson42XX.c */
```

Parameter Name	Data Type	I/O	Default	Min	Max
Vg	double	Input	2	-10	10
Vd1	double	Input	3	-10	10
Vd2	double	Input	5	-10	10

Each feature of the KULT interface is explained in the following sections.

Module identification area

The module identification area is directly below the menu bar and defines the presently open user library and user module. The components of this area are as follows:

- **Library**: Displays the name of the presently open (active) user library.
- **Module**: Displays the name of the presently open user module.
- **Return Type**: Defines the data type of all codes that are returned by `return(code)` statements in the user module. You can select one of the following variable types:
  - **char**: Character data
  - **double**: Double-precision data
  - **float**: Single-precision floating point data
  - **int**: Integer data
  - **long**: 32-bit integer data
  - **void**: No data returned

NOTE

When a user test module (UTM) is executed by Clarius, the value of the `return(code)` statement is displayed on the Data worksheet in the column labeled with the module name.

- **Library Visible / Library Hidden:** Displays whether or not the presently open user library is available to Clarius. To change the hidden or visible status, select or clear the Hide Library option in the Options menu (on page 1-15).
- **Apply:** Updates the presently open user module to reflect additions and changes.

Module parameter display area

The module parameter area is a display-only area that is directly below the module identification area. In the module-parameter area, KULT displays:

- The C-language function prototype for the user module, reflecting the parameters that are specified in the Parameters tab area, and the `return(code)` data type.
- The `#include` and `#define` statements that are specified in the **Includes** tab.

Module code-entry area

The module code-entry area is below the module-parameter area. The module code-entry area is where you enter, edit, or view the user-module C code. Scroll bars located to the right and below the module-code entry area let you move through the code.

NOTE

Do not enter the following C-code items in the module code-entry area (KULT enters these at special locations based on information in other places in KULT): `#include` and `#define` statements; the function prototype; and the terminating brace. To control internal or external instrumentation, use functions from the Linear Parametric Test Library (LPTLib). For more information, refer to *Model 4200A-SCS LPT Library Programming*.

Terminating brace area

The terminating-brace area is a display-only area. KULT automatically enters and displays the terminating brace for the user-module code when you select **Apply**.

Tab area

The Tab area includes the tabs:

- Parameters
- Includes
- Description
- Build

Parameters tab area

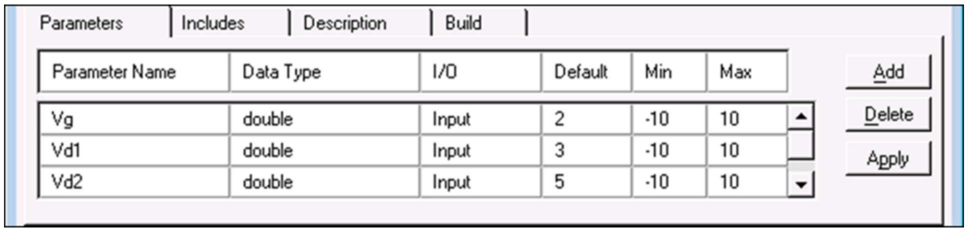
In the Parameters tab, you define and display parameters in the user module call. You can define and display:

- Parameter name
- Parameter data type
- Input or output (I/O) data direction
- Default, min, and max values for the parameter

These options are defined in the following text.

The Parameters tab area is near the bottom of the KULT main screen. An example is shown here.

Figure 3: Parameters tab for the Rdson42XX user module from the KI42XX library



Parameter Name	Data Type	I/O	Default	Min	Max
Vg	double	Input	2	-10	10
Vd1	double	Input	3	-10	10
Vd2	double	Input	5	-10	10

NOTE

You can right-click anywhere in the Parameters tab area to access the Add, Delete, and Apply options.

To add a parameter:

1. Select **Add**.
2. Enter the information as needed.
3. Select **Apply**.

To delete a parameter:

1. Select the parameter name or any of the adjacent fields.
2. Select **Delete**.

To make changes to the parameters:

1. Make changes in the appropriate field.
2. Select **Apply**.

Parameter name field

The parameter name field identifies the parameters that are passed to the user module. These are the same parameters that are specified in the user-module function prototype. KULT constructs the prototype from the Parameters tab entries when you select **Apply,** and then displays it in the module-parameter display area.

Data type field

The data type field specifies the parameter data type. Select the arrow at the right of the data type field to choose from a list of the following data types:

- **char:** Character data
- **char*:** Pointer to character data
- **float:** Single-precision floating point data
- **float*:** Pointer to single-precision floating point data
- **double:** Double-precision data
- **double*:** Pointer to double-precision point data
- **int:** Integer data
- **int*:** Pointer to integer data
- **long:** 32-bit integer data
- **long*:** Pointer to 32-bit integer data
- **F_ARRAY_T:** Floating point array type
- **I_ARRAY_T:** Integer array type
- **D_ARRAY_T:** Double-precision array type

I/O field

The I/O field defines whether the parameter is an input or output type. Select the arrow to the right of the I/O field to select from the input and output selections.

Default, min, and max fields

The **Default** field specifies the default value for a nonarray (only) input parameter.

The **Min** field specifies the minimum recommended value for a nonarray (only) input parameter. When the user module is used in a Clarius user test module (UTM), configuration of the UTM with a parameter value smaller than the minimum value causes Clarius to display an out-of-range message.

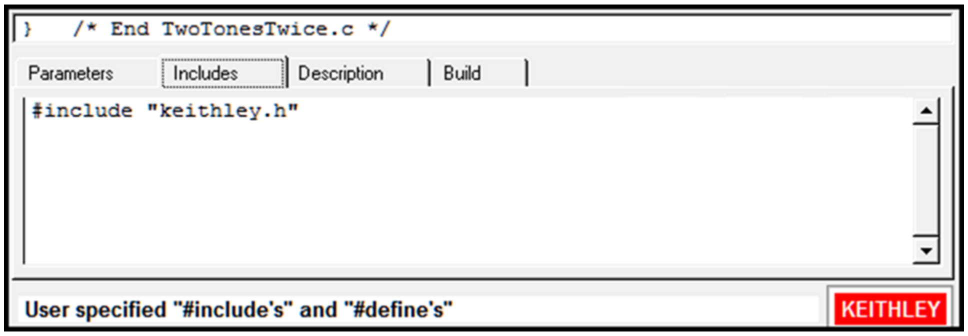
The **Max** field specifies the maximum recommended value for a nonarray (only) input parameter. When the user module is used in a Clarius UTM, configuration of the UTM with a parameter value larger than the maximum value causes Clarius to display an out-of-range message.

The minimum value must be less than the maximum value.

Includes tab area

The Includes tab, shown below, lists the header files used in the user module. This area can be used to add `#include` and `#define` statements to the presently open user module.

Figure 4: Default Includes tab area



```
}   /* End TwoTonesTwice.c */
```

```
#include "keithley.h"
```

User specified "#include's" and "#define's"

By default, KULT automatically enters the `keithley.h` header file into the Includes tab. The `keithley.h` header file includes the following frequently used C-programming interfaces:

- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <string.h>`
- `#include <math.h>`
- `#include "windows.h"`

In most cases, it is not necessary to add items to the Includes tab area, because `keithley.h` provides access to the most common C functions. However, in some cases, both of the following may apply:

- You do not want to include `keithley.h`
- You want to include only the header files specifically needed by your user module and all the user modules on which it depends.

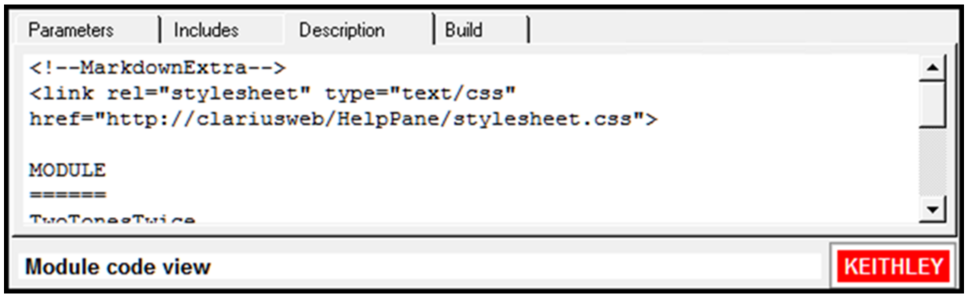
If so, you must minimally include the following header files and `#define` statements to properly build user modules and user libraries:

```
#include "lptdef.h"
#include "lptdef_lowercase.h"
#include "kilogmsg_proto.h
#include "ktemalloc.h"
#include "usrlib_proto.h"
#define PTexit _exit
#define exit Unsupported Syntax
#define abort Unsupported Syntax
#define terminate Unsupported Syntax
```

Description tab area

The Description tab, shown below, allows you to enter descriptive information for the presently open user module. The information that is entered in this area documents the module to the Clarius user and is used to create Clarius user library help.

Figure 5: Description tab area



```
<!--MarkdownExtra-->
<link rel="stylesheet" type="text/css"
href="http://clariusweb/HelpPane/stylesheet.css">

MODULE
======
TwoTonesTwice
```

CAUTION

Do not use C-code comment designators (/*, */, or //) in the Description tab area. When the user-module code is built, KULT also evaluates the text in this area. C-code comment designators in the Description tab area can be misinterpreted, causing errors.

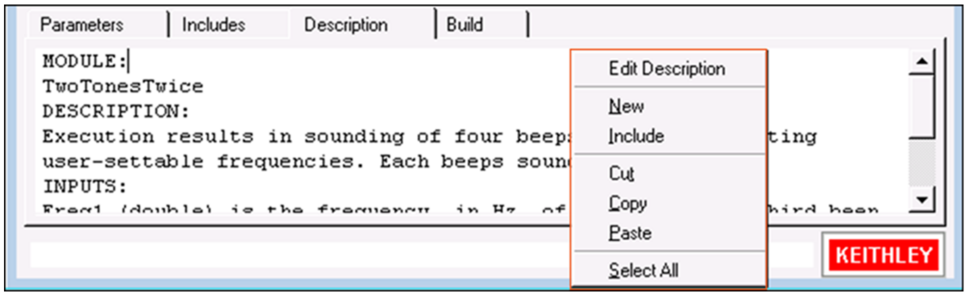
NOTE

Do not place a period in the first column (the left-most position) of any line in the Description tab area. Any text after a first-column period will not be displayed in the documentation area of a Clarius UTM definition document.

To enter a description:

1. Select in the **Description** tab area.
2. Enter the description.
3. Right-click in the **Description** tab area to open the menu shown here.

Figure 6: Edit menu for the Description tab area



```
MODULE:
TwoTonesTwice
DESCRIPTION:
Execution results in sounding of four beep         ting
user-settable frequencies. Each beeps soun
INPUTS:
Freq1 (double) is the frequency in Hz of           hird been
```

The edit menu commands are:

- **New**: Deletes the present description from the description tab area, allowing you to enter a new description.
- **Include**: Imports any file that you specify, typically a text file, into the document tab area. Refer to Include (on page 1-10) for more information.
- **Cut**: Removes highlighted text from the Description tab and copies it to the clipboard. The text on the clipboard can be restored to new locations, in or out of KULT, using the paste function.
- **Copy**: Copies highlighted text from the description tab area to the clipboard. The text on the clipboard can be placed at new locations, in or out of KULT, using the paste function.
- **Paste**: Places text from the clipboard at a selected location in the Description tab area.
- **Select All**: Selects everything in the Description tab area.

Include

Imports a `*.c` file that you specify into the module code-entry area only. This is typically a text file. The file is imported into the document tab area.

CAUTION

The File > Include **command inserts everything from the specified file. If the specified file is the source file for a KULT user module `<ModuleName.c>`, everything that KULT saves into the user module (not only the C code) is imported. Therefore, you must edit the entered text to remove all but the needed information. In particular, you must remove any comments of the form `/* USRLIB MODULE ___*/`.**

In some cases, it is more efficient to copy only the needed code text from the source file, then paste it into the module code-entry area.

NOTE

To insert a text or other file into the document tab area, refer to Description tab area (on page 1-9) for information about the Include menu option.

To import a *.c file:

1. Select **Include**. The Include Other File dialog opens.
2. Place the cursor where you want to place the new information.
3. Browse and select a file or enter a file name and path.
4. Select **Open**. The file is inserted at the cursor location.

Build tab area

The Build tab area displays any error or warning messages that are generated during a code build operation of the user library. When you select a build error message that is displayed in the Build tab area, KULT highlights either the line of code where the error occurred or the next line, depending on how the compiler caught the error. KULT also highlights the error message. This helps you correct errors.

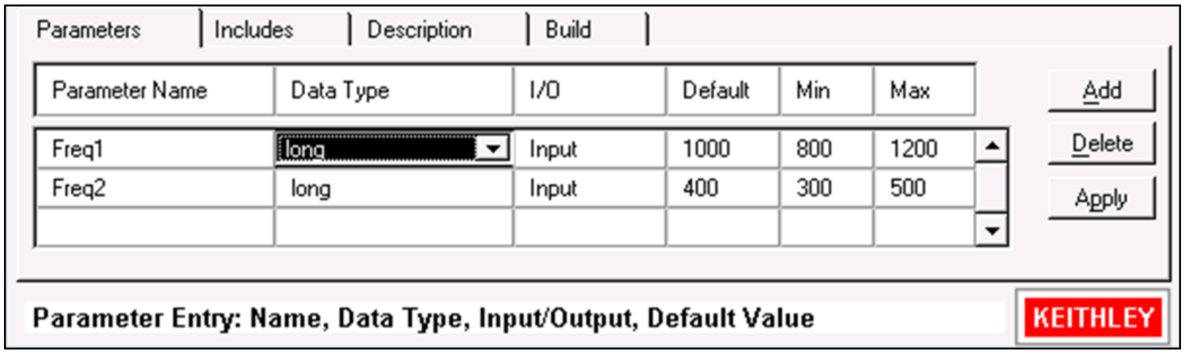
If no errors are found, the Build tab area displays:

```
No Errors/Warnings Reported. Compilation/Build was Successful.
```

Status bar

The status bar at the bottom of the KULT dialog displays a description of the area where the cursor is located. For example, if the cursor is in the Parameters tab area, the status bar describes that area, as shown in the following figure.

Figure 7: Example of description in status bar



Parameter Name	Data Type	I/O	Default	Min	Max
Freq1	long	Input	1000	800	1200
Freq2	long	Input	400	300	500

Parameter Entry: Name, Data Type, Input/Output, Default Value

Menus

This section describes the menus on the menu bar, which is at the top of the KULT dialog.

File menu

All user libraries are stored in the `C:\s4200\kiuser\usrlib` directory. This directory is referred to as Clarius/KULT user-library directory. It is the active user-library directory, which is where Clarius and KULT look for user libraries and user modules.

The File menu includes options to work with libraries.

New Library

The New Library menu option creates a new user library.

NOTE

Library names cannot start with a number.

To create a new user library:

1. Select **New Library**. The Enter library dialog opens.
2. Name the new user library.
3. Select **OK**. This initializes and opens the new user library in place of the presently open library.

Open Library

Opens an existing user library in place of the presently open library.

To open a library:

1. Select **Open Library** to display the open library list.
2. Select an existing user library.
3. Select **OK** to open the selected library.

Copy Library

Creates a copy of the presently open user library.

To copy a library:

1. Select **Copy Library**. The Enter Library dialog opens.
2. Name the new user library into which to copy the presently open library.
3. Select **OK** to copy the open user library into the new library.

Delete Library

Deletes an existing user library and all its contents.

To delete a library:

1. Select **Delete Library**. The list of libraries is displayed.
2. Select the user library to be deleted.
3. Select **OK** to delete the selected library.

New Module

This option creates a new user module. When you create a new user module, existing module information in the KULT interface is cleared.

The name of the new module must not duplicate the name of any existing user module or user library in the entire collection of user libraries.

To create a new user module:

1. Select **New Module**.
2. Enter a new user-module name in Module.
3. Select **Apply**.

Open Module

Opens an existing user module.

To open a module:

1. Select **Open Module**. The Open Module list is displayed.
2. Select an existing user module.
3. Select **OK** to open the selected module in place of the presently open module.

Save Module

Saves the open user module.

Copy Module

Creates a copy of the open user module.

The name of the new module must not duplicate the name of any existing user module or user library in the entire collection of user libraries.

To copy the user module:

1. Select **Copy Module**. The list of libraries opens.
2. Select the user library in which to copy the presently open user module.
3. Select **OK.** The Enter New Module dialog opens.
4. Enter a unique user-module name.
5. Select **OK.** The presently open module is copied into the selected library under the new name. The presently open module remains open.

Delete Module

Deletes a user module from the open user library.

To delete a user module:

1. Select **Delete Module**. The KULT: Library `[OpenLibraryName]` list is displayed.
2. Select the module to be deleted.
3. Select **OK**. The selected module is deleted. The open module continues to be displayed, even if it is the module that you deleted.

NOTE

The executable user-library file, a dynamic link library (DLL), contains the deleted module until you rebuild the library. Refer to Building the user library to include the new user module (on page 2-10) for more information.

Print Module

Prints a text file that contains all the information for the presently open user module. The text file is arranged in the form that KULT uses internally.

Exit

Exits KULT.

Edit menu

The **Edit** menu contains typical Microsoft® Windows® editing commands.

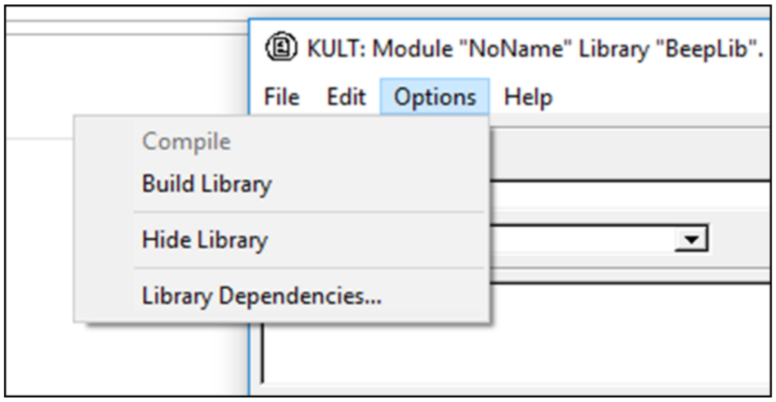
Edit menu commands:

- **Cut**: Removes highlighted text and copies it to the clipboard. The text on the clipboard can be restored to new locations, in or out of KULT, using the paste function.
- **Copy**: Copies highlighted text to the clipboard. The text on the clipboard can be placed at new locations, in or out of KULT, using the paste function.
- **Paste**: Places the text from the clipboard to a selected location.
- **Select All**: Selects everything in the module code-entry area.
- **Undo**: Allows you to reverse up to the last ten changes made in the module code-entry area.
- **Redo**: Allows you to reverse up to the last ten undo operations in the module code-entry area.

Options menu

The KULT Options menu is shown here.

Figure 8: KULT Options menu



Options menu commands:

- **Build Library:** When selected, adds the open user module (or updates changes) to the open user library. All the modules in the open user library and any libraries on which the open module depends are linked together. A dynamic link library (DLL) is created that is accessible using user test modules (UTMs) in Clarius.

NOTE

Some Keithley Instruments-supplied user libraries contain dependencies. If you need to build or rebuild such libraries, be sure that you specify the dependencies in the dialog opened by **Options > Library Dependencies**. For more information, refer to descriptions in the following and to details in the Dependent user modules and user libraries (on page 3-9).

Otherwise, the Build Library function will fail. For example, `ki82ulib` depends on `KI590ulib` and `Winulib`. You must specify these dependencies before rebuilding `ki82ulib` after making changes.

- **Hide Library:** When selected, causes the present user library to be unavailable to Clarius. For example, use Hide Library if you want to designate that a user library is only to be called by another user library and is not to be connected to a UTM.

- **Library Dependencies:** When selected, displays the Library Dependencies list, where you specify each user library that is called by and that must be linked to the open user library. You must make selections individually; do not hold down the control or shift key to make multiple selections.

NOTE

The `C:\s4200\kiuser\usrlib\<library name>\build` folder is created when you run the `bld_lib` subcommand or select the **Build Library** menu option. This folder can be safely deleted for debugging purposes.

Help menu

The Help menu contains online help information about KULT:

- **Contents:** Allows access to the online KULT manual and other 4200A-SCS reference information.
- **About KULT:** Displays the software version.

Develop and use user libraries

Clarius includes user libraries of user modules that contain precoded user modules for commonly used external instruments. You can use these as-is, customize them, or create new ones. Most user modules contain functions from the Keithley-supplied Linear Parametric Test Library (LPT Library) and ANSI-C functions. All user modules are created and built using KULT.

Additionally, using KULT, you can program custom user modules in C. The LPT Library contains functions that are designed for parametric tests. However, any C routine that can be built using KULT can be used as source code for a user module.

A user library is a dynamic link library (DLL) of user modules that are built and linked using the Keithley User Library Tool (KULT).

A user module is a C-language function that:

1. Typically calls functions from the LPT library and ANSI-C functions.
2. Is developed using the Keithley User Library Tool (KULT).

The default collection of KULT user libraries is stored in the directory `C:\s4200\kiuser\usrlib`.

NOTE

User library names must not start with a number.

Copy user modules and files

You can use the KULT zip (on page 3-8) and unzip (on page 3-9) subcommands to copy user libraries and other files. See Performing other KULT tasks using command-line commands (on page 3-3) for more information.

The `KULTArchive.exe` utility is installed on your 4200A-SCS. You can copy this utility to a Model 4200 or 4200A-SCS to archive or unzip a user library for use with an earlier version of Clarius. This utility is located at `C:\S4200\sys\bin\KULTArchive.exe`.

If you use the `KULTArchive.exe` utility with a Model 4200, you must install the Microsoft Visual C++ Redistributable. This file is available on your 4200A-SCS at `C:\s4200\sys\Microsoft\Microsoft Visual C++ 2017 Redistributable\c_redistx86.exe`.

Usage

```
kultarchive [subcommand]
```

Where:

`<subcommand>` is the zip or unzip operation.

KULTArchive zip subcommand

```
zip -l<library_name> [password] <zipfile_name>
```

The `<library_name>` user library is created in the active user-library directory.

The `[password]` parameter is optional.

Example for zip without password

```
kultarchive zip -l<Library1> C:\temp\myzip.zip
```

KULTArchive unzip subcommand

```
unzip [-dest_path] [password] <zipfile_name>
```

Where:

- `[-dest_path]` is the target directory where the file will be unzipped.
- `[password]` is required if the file was compressed using the password parameter in the `zip` subcommand.

The `<`*`zipfile_name`*`>` archive is unzipped in the active user-library directory unless the `[-dest_path]` parameter is specified. The `[-dest_path]` parameter should not be used when you import a user library.

Example for unzip with password

```
kultarchive unzip -password -pw1234 C:\temp\myzip.zip
```

Enabling real-time plotting for UTMs

To enable real-time plotting in a UTM, you use the following LPT library functions:

- `PostDataDouble()`
- `PostDataInt()`
- `PostDataString()`

In these functions, the first parameter is the variable name, defined as `char *`.

When using the new functions to transfer data into the data sheet in real time, make sure the data is already in the memory of the 4200A-SCS. Sweep measurements are not suitable for real-time transfer because data is not ready until sweep finishes. The following tutorials show how to enable real-time plotting for a UTM.

For more information on LPT library functions, refer to *Model 4200A-SCS LPT Library Programming*.

Using NI-VISA in user libraries

You can use a user library to communicate with an external instrument that is connected using a USB cable. The library requires the optional NI-VISA installation. To include NI-VISA, a library dependency to `visa32.lib` must be added first. This dependency applies to all modules in a library and only needs to be completed once per library.

Clarius includes two libraries, `generic_visa_ulib` and the `dmm_6500_7510_temp_ulib`, as examples of using VISA commands to communicate with USB controlled instruments.

Add NI-VISA as a library dependency in KULT

To add NI-VISA as a library dependency in KULT:

1. Close KULT.
2. Go to the `kitt_src` folder for the library, such as `C:\s4200\kiuser\BeepLib\lib_name\kitt_src`.
3. Open the `.mak` file for the library in Notepad or another editor.
4. In the LIBS variable, between the quotes, enter `visa32.lib`. Enter any other library dependencies you may need.
5. Save the file.
6. Reopen the library in KULT.

NOTE

Modifying the library dependencies in KULT will overwrite NI-VISA. To add additional dependencies without overwriting VISA, repeat the above process.

Add NI-VISA as a library dependency in the KULT Extension

In addition to the library dependency, all modules that use NI-VISA must also include the `visa.h` and `visatype.h` header files.

To add NI-VISA as a library dependency in the KULT Extension:

1. Select the library in the KULT side bar.
2. In the Miscellaneous pane of the KULT side bar, select the *`library_name`*`.mak` file to open it in the editor.
3. In the code editor, add the `visa32.lib` file to the LIBS variable.
4. Save the file.

Include the NI-VISA header files in KULT

To include the NI-VISA header files in KULT:

1. Open the module in KULT.
2. Select the Includes tab at the bottom of the screen.
3. Add the following statements:
```
#include "visa.h"
#include "visatype.h"
```

Include the NI-VISA header files in the KULT Extension

To include the NI-VISA header files in the KULT Extension:

1. Open the module in the editor.
2. Under the `/* USRLIB MODULE PARAMETER LIST */` comment, add the following statements:
   ```
   #include "visa.h"
   #include "visatype.h"
   ```

Remove Intellisense errors

If you are using the KULT Extension, including `visa.h` and `visatype.h` may cause an Intellisense error, because the Intellisense configuration file cannot find the path to the header files. This error will not affect building the library, but you can remove it by editing the `c_cpp_properties.json` file.

To remove Intellisense errors caused by the NI-VISA header files:

1. Open the `c_cpp_properties.json` header file from the Miscellaneous pane of the KULT side bar.
2. In the editor, add the path to the header files:
   ```
   C:/Program Files (x86)/IVI Foundation/VISA/WinNT/include
   ```
   Included paths should be enclosed in double quotes and separated by commas.
3. Save the file. This applies to all libraries in the working directory of Visual Studio Code.

NI-VISA commands must be used to communicate with the instrument. These commands are documented in the *NI-VISA Programmer Reference Manual*. The most commonly used commands are shown in the following table.

Commonly used VISA commands

Command Name	Description
`viOpenDefaultRM`	Initializes VISA. Must be called before any other VISA command.
`viFindRsrc`	Finds available instruments and returns a list of their resource strings. The list can be filtered to USB only using the format string `USB?*`
`viFindNext`	Used to iterate through the returned list of instruments from viFindRsrc to find an instrument.
`viOpen`	Opens a session to the instrument specified by the VISA resource string.
`viWrite`	Writes data to an external instrument.
`viRead`	Reads a set number of characters as a string from the output buffer of the external instrument.
`viClose`	Closes a VISA session. Use this command before exiting a user module.

For more information on VISA command syntax, usage, and error codes, refer to the *NI-VISA Programmer Reference Manual*, available at ni.com/.

Section 2

KULT tutorials

In this section:

KULT Tutorials

The tutorials in this section provide step-by-step instructions for accomplishing common tasks with KULT. The tutorials are summarized here.

***Tutorial: Creating a new user library and new user module* (on page 2-3)**

- Name a new user library
- Name a new user module
- Enter a return type
- Enter user module code
- Enter parameters
- Enter header files
- Document the user module
- Save the user module
- Build the user module
- Find code errors
- Build the user library to include the new user module
- Find build errors
- Check the user module

***Tutorial: Creating a user module that returns data arrays* (on page 2-14)**

- Name a new user library and new `VSweep` user module
- Enter the `VSweep` user-module return type
- Enter the `VSweep` user-module code
- Enter the `VSweep` user-module parameters
- Enter the `VSweep` user-module header files
- Document the `VSweep` user module
- Save the `VSweep` user module
- Build the `VSweep` user module
- Check the `VSweep` user module

***Tutorial: Creating a user module that returns data arrays in real time* (on page 2-20)**

- Name a new user library and new `VSweepRT` user module
- Enter the `VSweepRT` user-module return type
- Enter the `VSweepRT` user-module code
- Enter the `VSweepRT` user-module parameters
- Enter the `VSweepRT` user-module header files
- Document the `VSweepRT` user module
- Save the `VSweepRT` user module
- Build the `VSweepRT` user module
- Check the `VSweepRT` user module

***Tutorial: Calling one user module from within another* (on page 2-25)**

- Create the `VSweepBeep` user module by copying an existing user module
- Call an independent user module from the `VSweepBeep` user module
- Specify user library dependencies in the `VSweepBeep` user module
- Build the `VSweepBeep` user module
- Check the `VSweepBeep` user module

***Tutorial: Customizing a user test module (UTM)* (on page 2-30)**

- Copy the `Rdson42XX` user module to `RdsonAvg`
- Modify the `RdsonAvg` user module
- Change a parameter name
- Change the module description
- Save and build the library
- Add `RdsonAvg` to the `ivswitch` project

***Tutorial: Creating a user module for stepping or sweeping* (on page 2-38)**

- Name a new user library and new `vds_id_step_sweep` user module
- Enter the `vds_id_step_sweep` user-module return type
- Enter the `vds_id_step_sweep` user-module code
- Enter the `vds_id_step_sweep` user-module parameters
- Enter the `vds_id_step_sweep` user-module header files
- Document the `vds_id_step_sweep` user module
- Save the `vds_id_step_sweep` user module
- Build the `vds_id_step_sweep` user module
- Set up the user interface of the `vds_id_step_sweep` user module
- Check the `vds_id_step_sweep` user module in Clarius

Tutorial: Creating a new user library and user module

KULT is a tool that helps you develop user libraries. Each user library is comprised of one or more user modules. Each user module is created using the C programming language.

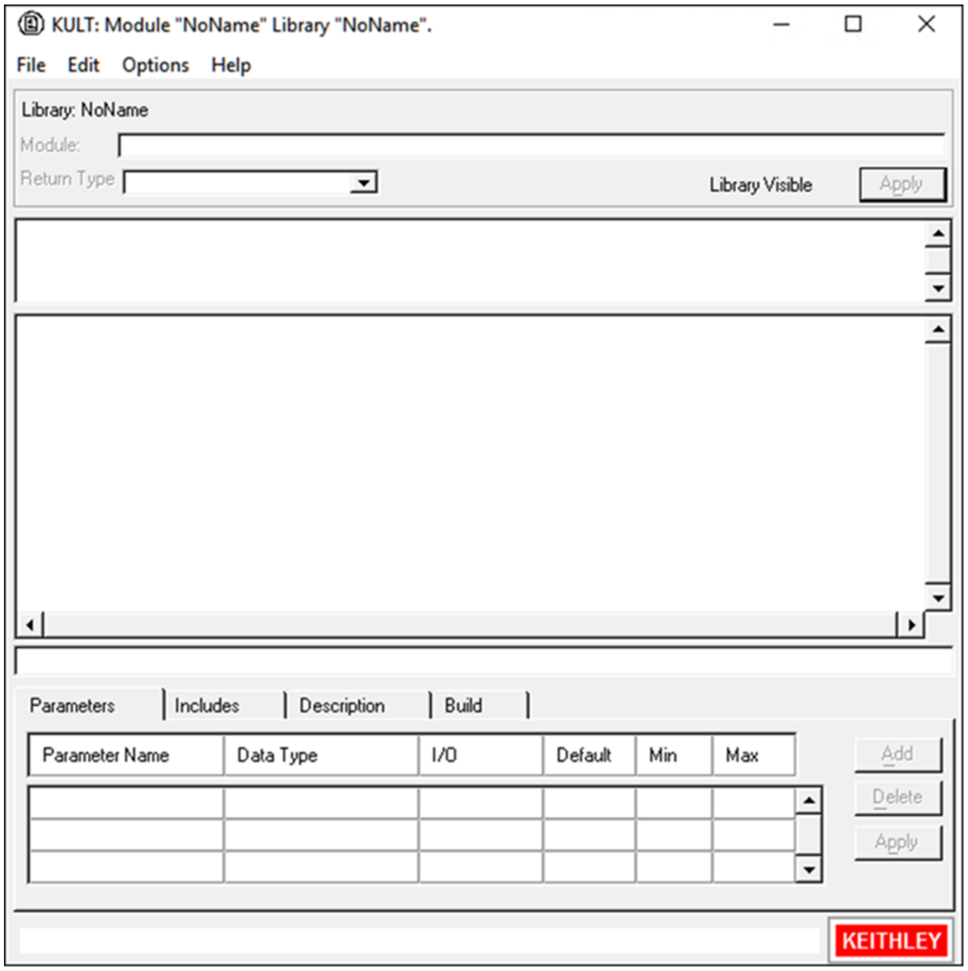
This section contains a tutorial that shows you how to create a new user library and new user module. A hands-on example is provided that illustrates how to create a user library that contains a user module that activates the internal beeper of the 4200A-SCS.

Starting KULT

To start KULT:

1. Select **KULT** in the Microsoft® Windows® **Start** menu (**Start > Keithley Instruments > KULT**).
2. A blank KULT dialog appears named **KULT:** Module **"NoName"** Library **"NoName"**, as shown in the following figure.

Figure 9: Blank KULT dialog



Naming a new user library

NOTE

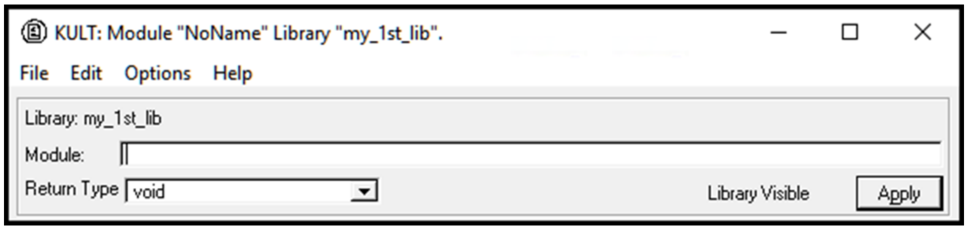
User library names cannot start with a number and cannot contain spaces.

To name a new user library:

1. In KULT, select **File > New Library**.
2. Enter the new user library name. For this tutorial, enter `my_1st_lib`.
3. Select **OK**.

   The dialog name changes to `KULT: Module "NoName" Library "my_1st_lib"`, and the name next to library in the top left of the dialog is now `my_1st_lib`, as shown in the following figure.

Figure 10: KULT after naming a user library



Creating a new user module

NOTE

When naming a user module, conform to case-sensitive C programming language naming conventions. Do not duplicate names of existing user modules or user libraries.

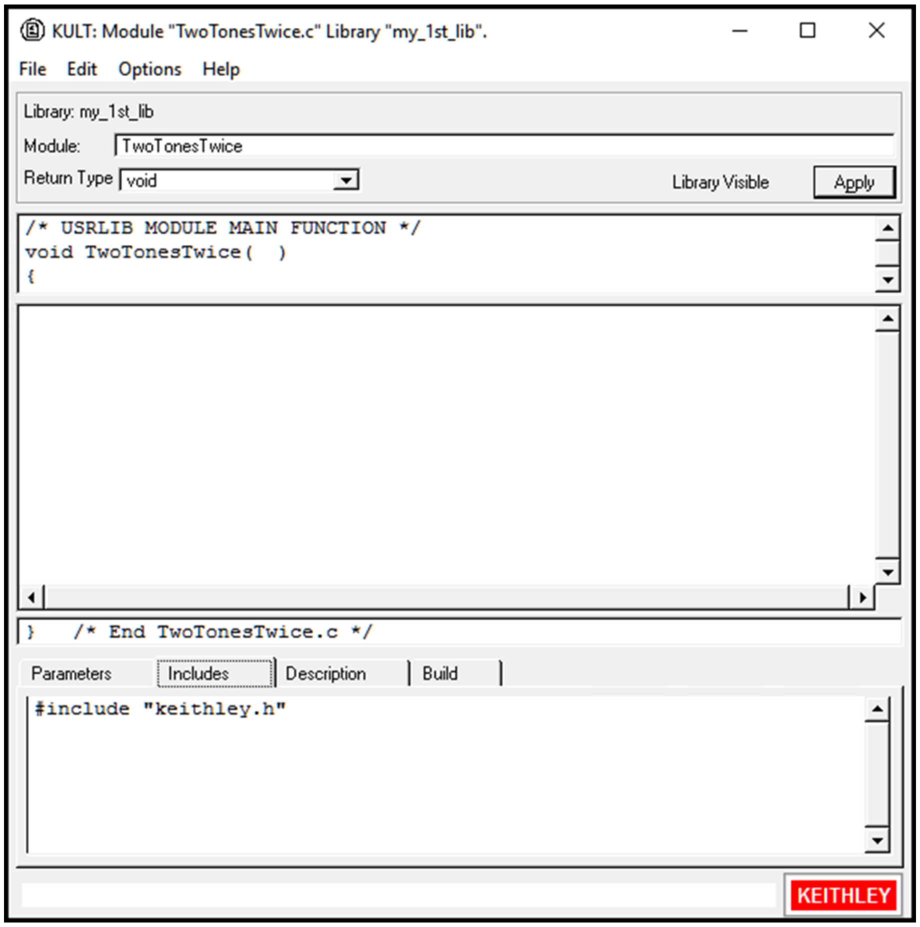
To create a new user module:

1. Select **File > New Module.**
2. In the Module text box at the top of the KULT dialog, enter the new user module name. For this tutorial, enter `TwoTonesTwice` as the new user module name.
3. Select **Apply**.

The KULT dialog changes as follows:

- The name of the dialog changes to `KULT: Module "TwoTonesTwice.c" Library "my_1st_lib"`.
- You see entries in the user-module parameters display area and in the terminating-brace display. If you select the **Includes** tab, there is also an entry there, as shown in the following figure.

Figure 11: KULT after naming a user module



```
/* USRLIB MODULE MAIN FUNCTION */
void TwoTonesTwice(  )
{
```

```
}   /* End TwoTonesTwice.c */
```

```
#include "keithley.h"
```

NOTE

To view the entire module parameter display area, use the scroll bar.

Entering the return type

If your user module generates a return value, select the data type for the return value in the **Return Type** box. The `TwoTonesTwice` user module does not produce a return value, so keep the `void` default entry.

Entering user module code

Enter the C code into the module-code entry area.

NOTE

Refer to *Model 4200A-SCS LPT Library Programming* for a complete list of supported I/O and SMU commands.

For the `TwoTonesTwice` user module, enter the code listed below. The code deliberately contains a missing `;` error to illustrate KULT debug capability.

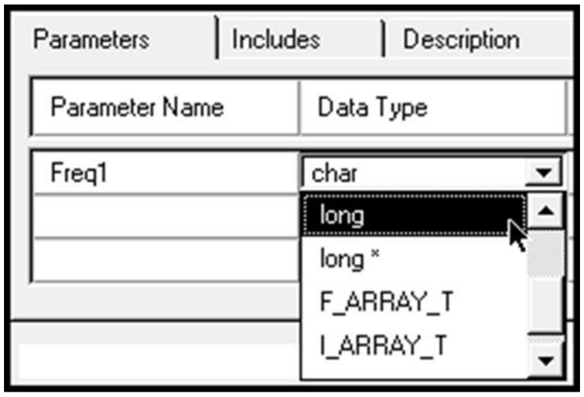
```
/* Beeps four times at two alternating user-settable frequencies. */
/* Makes use of Windows Beep (frequency, duration) function. */
/* Frequency of beep is long integer, in units of Hz. */
/* Duration of beep is long integer, in units of milliseconds. */
Beep(Freq1, 500);  /* Beep at first frequency for 500 ms */
Beep(Freq2, 500);  /* Beep at second frequency */
Beep(Freq1, 500);
Beep(Freq2, 500);
Sleep(500)        /* NOTE deliberately leave out semicolon */
```

Entering parameters

To enter the required parameters for the code:

1. Select the **Parameters** tab.
2. Select **Add** at the right side of the parameters tab area.
3. Under **Parameter Name**, enter `Freq1`.
4. Select the Data Type cell and select **long**, as shown here. This is the C data type.

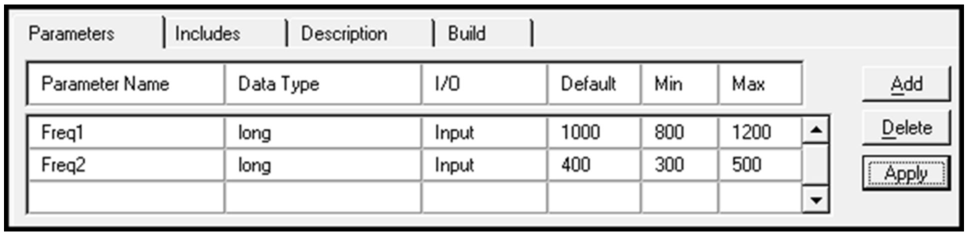
Figure 12: Data Type menu



5. For this user module, the `I/O` selection of `Input` is correct. If the Data Type is a pointer or array, you could choose `Input` or `Output`.

6. Under **Default**, **Min**, and **Max**, enter default, minimum, and maximum values. These values limit the choices the user sees. For the `TwoTonesTwice` user module, enter `1000`, `800`, and `1200`, respectively.
7. For the `TwoTonesTwice` module, add one more parameter with the values:
   - **Parameter name:** `Freq2`
   - **Data type:** `long`
   - **I/O:** `Input`
   - **Default:** `400`
   - **Min:** `300`
   - **Max:** `500`
8. Select **Apply**. (The Apply buttons at the top and bottom of the dialog act identically.)

Figure 13: Parameter entries for the TwoTonesTwice user module



Parameter Name	Data Type	I/O	Default	Min	Max
Freq1	long	Input	1000	800	1200
Freq2	long	Input	400	300	500

NOTE

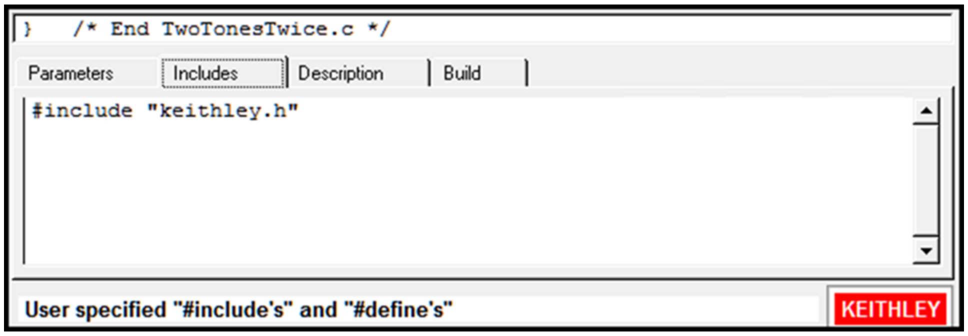
For an output parameter, only the following data types are acceptable: pointers (such as char*, float*, and double) and arrays (`I_ARRAY_T`, `F_ARRAY_T`, or `D_ARRAY_T`).

Entering header files

To enter the header files:

1. Select the **Includes** tab at the bottom of the dialog.

Figure 14: Default Includes tab area



```
}   /* End TwoTonesTwice.c */
```

```
#include "keithley.h"
```

User specified "#include's" and "#define's"

2. Enter any additional header files that are needed by the user module. No additional header files are needed for the `TwoTonesTwice` user module or for any of the user libraries supplied by Keithley Instruments.
3. Select **Apply**.

Documenting the user module

To document the user module:

1. Select the **Description** tab at the bottom of the dialog.
2. Enter any text needed to adequately document the user module to the Clarius user.

CAUTION

Do not use C-code comment designators (/*, */, or //) in the Description tab area. When the user-module code is built, KULT also evaluates the text in this area. C-code comment designators in the Description tab area can be misinterpreted, causing errors.

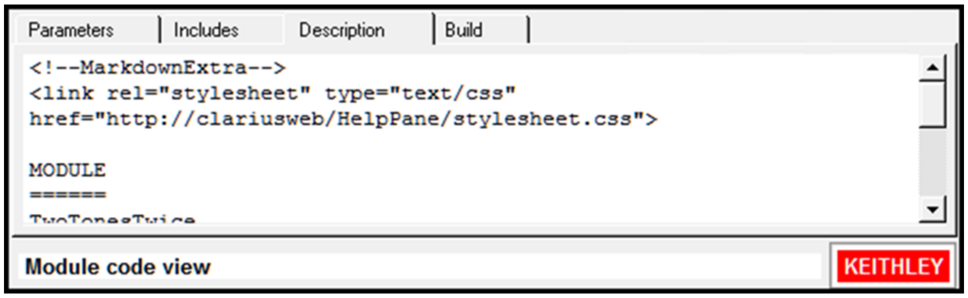
NOTE

Do not place a period in the first column (the left-most position) of any line in the Description tab area. Any text after a first-column period will not be displayed in the documentation area of a Clarius UTM definition document.

3. For the `TwoTonesTwice` user module, copy the following information into the **Description** tab:

```
<!--MarkdownExtra-->
<link rel="stylesheet" type="text/css"
   href="http://clariusweb/HelpPane/stylesheet.css">
MODULE
======
TwoTonesTwice

DESCRIPTION
-----------
Execution results in sounding of four beeps at two alternating user-settable
   frequencies. Each beeps sounds for 500 ms.

INPUTS
------
Freq1 (double) is the frequency, in Hz, of the first and third beep.
Freq2 (double) is the frequency, in Hz, of the second and fourth beep.

OUTPUTS
-------
None

RETURN VALUES
-------------
None
```

Figure 15: Description tab area



```
<!--MarkdownExtra-->
<link rel="stylesheet" type="text/css"
href="http://clariusweb/HelpPane/stylesheet.css">

MODULE
======
```

Saving the user module

Select the **File** menu, then select **Save Module**.

Building the user library to include the new user module

Build the user library to include the module.

To build the user library:

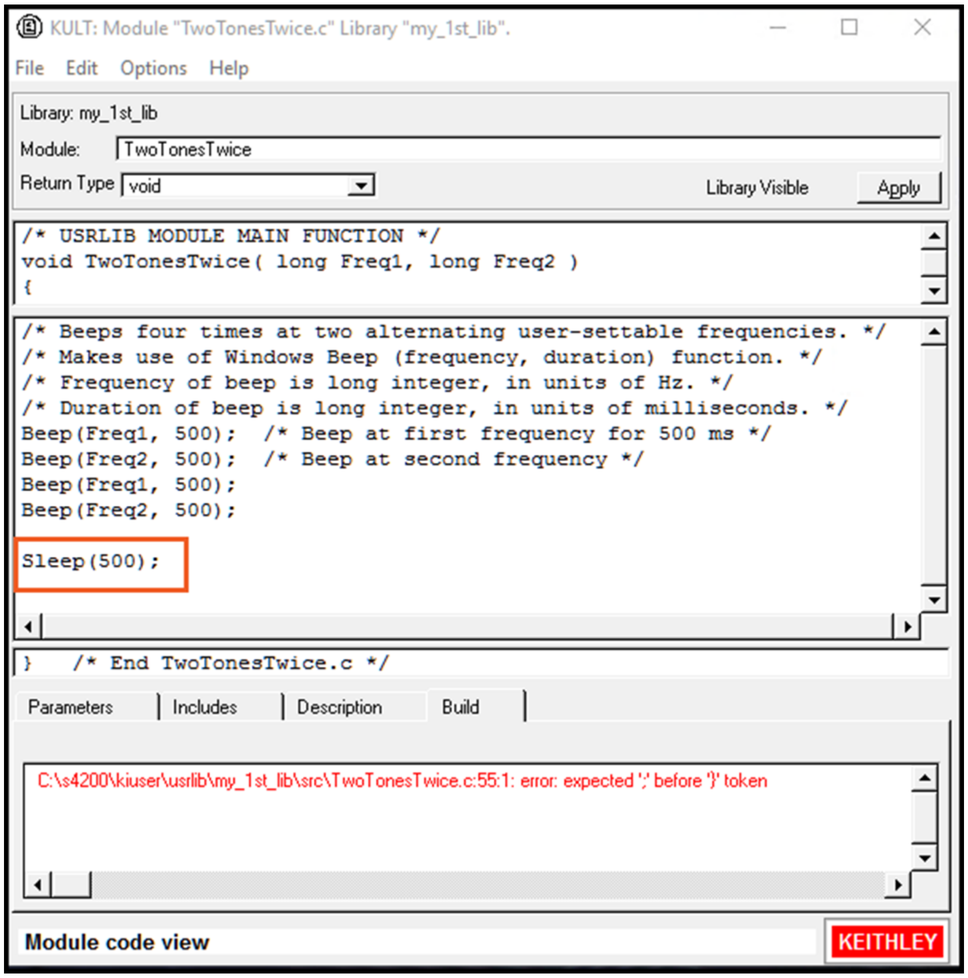
1. Select the Build tab.
2. From the **Options** menu, select **Build Library**. The following occurs:
   - The user library is built. All the user modules in the presently open user library and any libraries on which the presently open user module depends are linked together.
   - A DLL is created that is accessible using UTMs in Clarius.
   - The KULT Build Library message box indicates the build progress. If problems are encountered, this message box displays error messages. When you build the `TwoTonesTwice` user module, you should see an error.

Finding build errors

To find code errors for the `TwoTonesTwice` user module:

1. Review the error in the Build tab.

Figure 16: Find a code error



```
/* USRLIB MODULE MAIN FUNCTION */
void TwoTonesTwice( long Freq1, long Freq2 )
{
```

```
/* Beeps four times at two alternating user-settable frequencies. */
/* Makes use of Windows Beep (frequency, duration) function. */
/* Frequency of beep is long integer, in units of Hz. */
/* Duration of beep is long integer, in units of milliseconds. */
Beep(Freq1, 500);  /* Beep at first frequency for 500 ms */
Beep(Freq2, 500);  /* Beep at second frequency */
Beep(Freq1, 500);
Beep(Freq2, 500);

Sleep(500);
```

```
}   /* End TwoTonesTwice.c */
```

```
C:\s4200\kiuser\usrlib\my_1st_lib\src\TwoTonesTwice.c:55:1: error: expected ';' before '}' token
```

2. Add the missing semicolon at the end of the code [`Sleep(500);`] and delete the comment about the missing semicolon.
3. Select **File > Save Module**.
4. Select **Options > Build Library**.
   - The KULT Build message box should now display no error messages.
   - The Build tab area should display “No errors or warnings reported: Library was successfully built.”

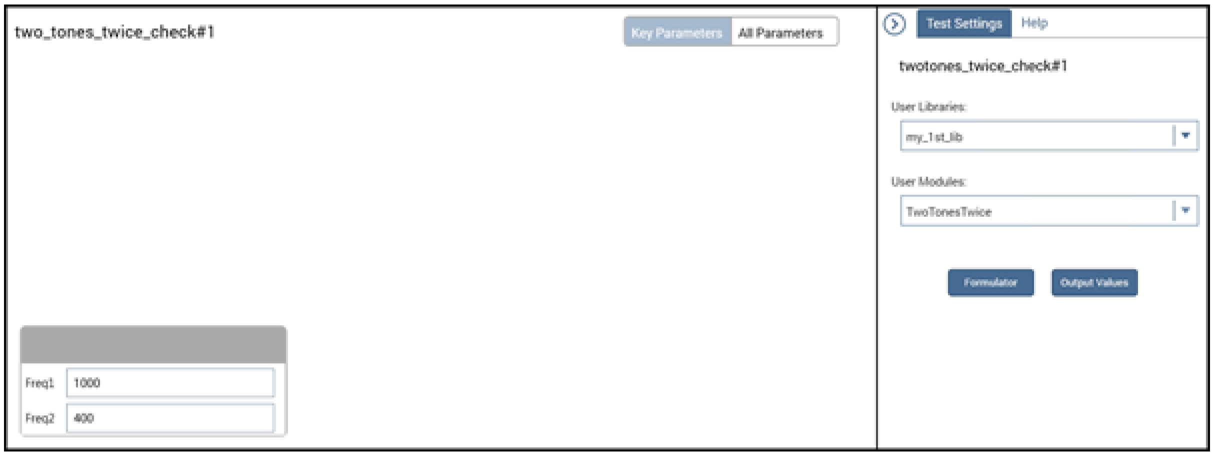
Checking the user module

To check a user module, you need to create and execute a user test module (UTM) in Clarius. Create a simple Clarius project to check the user module.

To check the user module in Clarius:

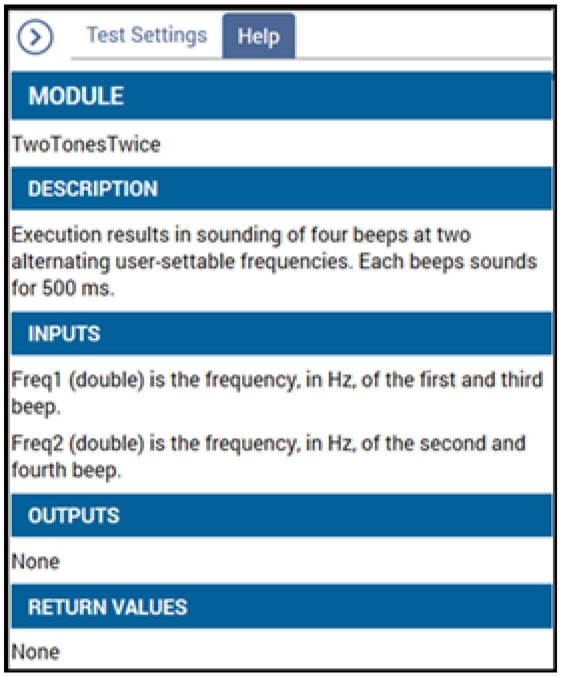
1. Start Clarius. If Clarius is already running, restart it.
2. Choose the **Select** pane.
3. Select the **Projects** tab.
4. Select **New Project**.
5. Select **Create**. You are prompted to replace the existing project.
6. Select **Yes**.
7. Select **Rename**.
8. Enter `UserModCheck` and press **Enter**.
9. Choose **Select**.
10. Select the **Actions** tab.
11. Drag **Custom Action** to the project tree. The action has a red triangle next to it to indicate that it is not configured.
12. Select **Rename**.
13. Enter `2tones_twice_chk` and press **Enter**.
14. Select **Configure**.
15. In the Test Settings pane, select the `my_1st_lib` user library.
16. From the User Modules list, select the `TwoTonesTwice` user module. A group of parameters are displayed for the UTM as shown in the following figure. Accept the default parameters for now. You can experiment later after you establish that the user module executes correctly.

Figure 17: Configured UTM



17. Select Help to verify that the HTML in the Description tab is correctly formatted. An example is shown in the following figure.

Figure 18: Example of help formatted as HTML for a user module



MODULE

TwoTonesTwice

DESCRIPTION

Execution results in sounding of four beeps at two alternating user-settable frequencies. Each beeps sounds for 500 ms.

INPUTS

Freq1 (double) is the frequency, in Hz, of the first and third beep.

Freq2 (double) is the frequency, in Hz, of the second and fourth beep.

OUTPUTS

None

RETURN VALUES

None

18. Select **Save**.
19. Execute the UTM by selecting **Run**. You should hear a sequence of four tones, sounded at alternating frequencies.

This tutorial generates no data. For an example of numerical data, see Tutorial: Creating a user module that returns data arrays (on page 2-14).

Tutorial: Creating a user module that returns data arrays

This section provides a tutorial that helps you use array variables in KULT. It also illustrates the use of return types (or codes), and the use of two functions from the Keithley Linear Parametric Test Library (LPTLib).

NOTE

Most of the basic steps that were detailed in Tutorial: Creating a new user library and user module (on page 2-3) are abbreviated in this tutorial.

Naming new user library and new VSweep user module

To name new user library and new VSweep user module:

1. Start **KULT**.
2. Select **File > New Library**.
3. In the Enter Library dialog that appears, enter `my_2nd_lib` as the new user library name.
4. Select **OK**.
5. Select **File > New Module**.
6. In the Module text box at the top of the KULT dialog, enter `VSweep` as the new module name.
7. Select **Apply**.

Entering the VSweep user-module return type

Select **int** from the Return Type list. This configures the VSweep user module to generate an integer return value.

Entering the VSweep user-module code

In the module code-entry area, enter the C code below for the VSweep user module. Open the KULT dialog to full screen view to simplify code entry.

```
/* VSweep module
   --------------
Sweeps through specified V range & measures I, using specified number of points.
Places forced voltage & measured current values (Vforce and Imeas) in output
   arrays.
NOTE For n increments, specify n+1 array size (for both NumIPoints and NumVPoints).

*/
double vstep, v; /* Declaration of module internal variables. */
int i;
if ( (Vstart == Vstop) )       /* Stops execution and returns -1 if */
    return( -1 );       /* sweep range is zero.              */

if ( (NumIPoints != NumVPoints) )    /* Stops execution and returns -2 if */
    return( -2 );   /* V and I array sizes do not match.  */

vstep = (Vstop-Vstart) / (NumVPoints -1);   /* Calculates V-increment size. */

for(i=0, v = Vstart; i < NumIPoints; i++)   /* Loops through specified number of */
   /* points. */
{
forcev(SMU1, v);   /* LPTLib function forceX, which forces a V or I.  */
measi(SMU1, &Imeas[i]);   /* LPTLib function measX, which measures a V or I. */
   /* Be sure to specify the *address* of the array. */

Vforce[i] = v;   /* Returns Vforce array for display in UTM Sheet. */

v = v + vstep;   /* Increments the forced voltage. */
 }

return( 0 );   /* Returns zero if execution Ok.*/
```

Entering the VSweep user-module parameters

This example uses the double-precision `D_ARRAY_T` array type. The `D_ARRAY_T`, `I_ARRAY_T`, and `F_ARRAY_T` are special array types that are unique to KULT. For each of these array types, you cannot enter values in the Default, Min, and Max fields. On the scroll bar in the Parameters tab area, there is a space below the slider. This space indicates a hidden fourth line of incomplete parameter information for the array-size parameter specification.

NOTE

When executing the Vsweep user module in a UTM, the start and stop voltages (Vstart and Vstop) must differ. Otherwise, the first return statement in the code halts execution and returns an error number (-1). When a user module is executed using a Clarius UTM, this return code is stored in the UTM Data worksheet. The return code is stored in a column that is labeled with the user-module name.

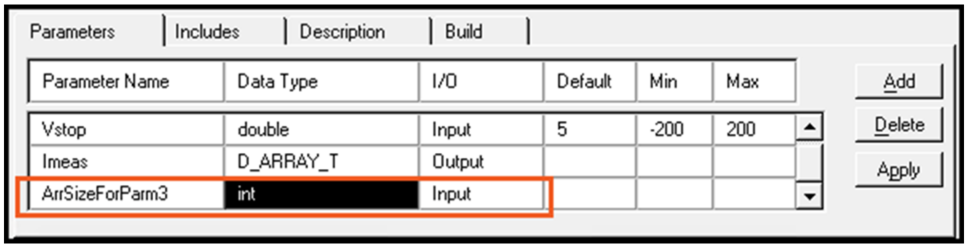
To enter the required parameters for the code:

1. Select the **Parameters** tab.
2. Enter the information for the two voltage input parameters, as shown in the following table. Select the **Add** button before adding each new parameter.

Parameter name	**Data type**	**I/O**	**Default**	**Min**	**Max**
Vstart	double	Input	0	−200	200
Vstop	double	Input	5	−200	200

3. Select **Add**.
4. Enter the following measured-current parameter information:
   - Parameter Name: `Imeas`
   - Data type: `D_ARRAY_T`
   - I/O: Output
5. Scroll down to display line 4 of the Parameters tab area. KULT enters the array size parameter in this line automatically for the array that is specified on line 3, as shown in the following figure.

Figure 19: KULT-entered array-size parameters



Parameter Name	Data Type	I/O	Default	Min	Max
Vstop	double	Input	5	-200	200
Imeas	D_ARRAY_T	Output			
ArrSizeForParm3	int	Input			

6. Under Parameter Name, change `ArrSizeForParm3` to `NumIPoints`. The default Parameter Name entry is only a description of the required array size parameter. You must replace it with an appropriate array size parameter, as required by the user module code.
7. Leave the Data Type and I/O entries as is.
8. Under Default, enter the number `11` for the default current-array size. You can also add Min and Max array sizes if needed.
9. Select **Add**.
10. Enter the following forced-voltage parameter information:
    - Parameter Name: `Vforce`
    - Data type: `D_ARRAY_T`
    - I/O: Output
11. Under **Parameter Name**, change `ArrSizeForParm5` to `NumVPoints`.

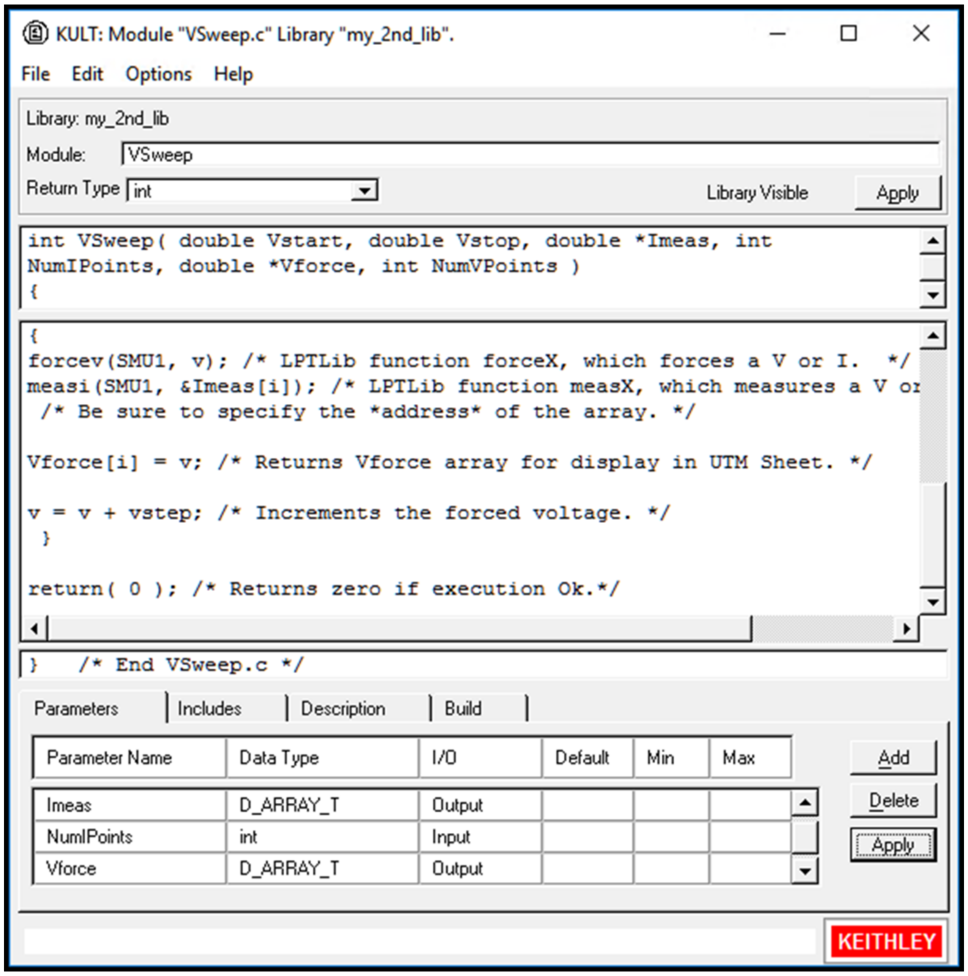
12. Under **Default**, enter the number `11` for the default voltage array size.

NOTE

When executing the `VSweep` user module in a UTM, the current and voltage array sizes must match; `NumIPoints` must equal `NumVPoints`. If the sizes do not match, the second return statement in the code halts execution and returns an error number (`-2`) in the `VSweep` column of the UTM Data worksheet.

13. Select **Apply.** In the module-parameter display area, the function prototype now includes the declared parameters, as shown in the following figure.

Figure 20: VSweep user-module dialog after entering and applying code and parameters


KULT: Module "VSweep.c" Library "my_2nd_lib".
File Edit Options Help
Library Visible
Apply
Parameters Includes Description Build
Add
Delete
Apply
KEITHLEY

Library: my_2nd_lib

Module: VSweep

Return Type: int

```
int VSweep( double Vstart, double Vstop, double *Imeas, int
NumIPoints, double *Vforce, int NumVPoints )
{
```

```
{
forcev(SMU1, v); /* LPTLib function forceX, which forces a V or I.  */
measi(SMU1, &Imeas[i]); /* LPTLib function measX, which measures a V or
 /* Be sure to specify the *address* of the array. */

Vforce[i] = v; /* Returns Vforce array for display in UTM Sheet. */

v = v + vstep; /* Increments the forced voltage. */
 }

return( 0 ); /* Returns zero if execution Ok.*/
```

```
}   /* End VSweep.c */
```

Parameter Name	Data Type	I/O	Default	Min	Max
Imeas	D_ARRAY_T	Output			
NumIPoints	int	Input			
Vforce	D_ARRAY_T	Output			

Entering the VSweep user-module header files

You do not need to enter any header files for the VSweep user module. The default `keithley.h` header file is sufficient.

Documenting the VSweep user module

Select the Description tab and enter documentation for the user module, based on the comments provided in the code and other information about the module.

Saving the VSweep user module

From the **File** menu, select **Save Module**.

Building the VSweep user module

To build the user module:

1. Select the **Build** tab at the bottom of the dialog to open the Build tab area.
2. In the **Options** menu, select **Build Library**. The user library builds. You should not see error messages.

NOTE

If you do see error messages, check for typographic errors, then fix and rebuild the user module. If necessary, review Finding build errors (on page 2-11).

Checking the VSweep user module

Check the user module by creating and executing a UTM in Clarius.

To check the user module:

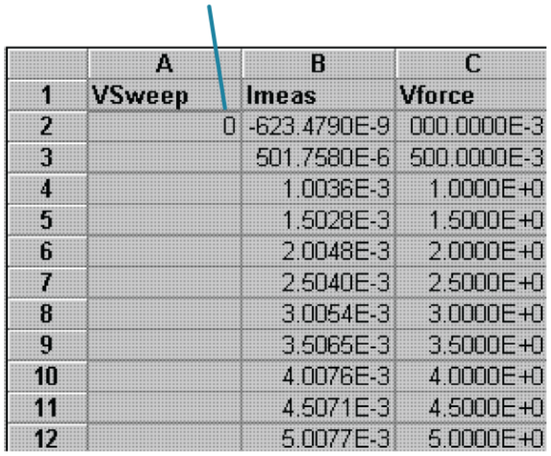
1. Connect a 1 kΩ resistor between the FORCE terminal of the ground unit (GNDU) and the FORCE terminal of SMU1.
2. Instead of creating a new project, reuse the `UserModCheck` project that you created in Tutorial: Creating a new user library and user module (on page 2-3).
3. Choose **Select**.
4. Select the **Devices** tab.
5. Select the 2-wire-resistor.
6. Choose **Select**.
7. Select the **Tests** tab.

8. For the Custom Test, select **Choose a test from the pre-programmed library (UTM)**.
9. Drag **Custom Test** to the project tree. The test has a red triangle next to it to indicate that it is not configured.
10. Select **Rename**.
11. Enter the name `v_sweep_chk`. You will use this UTM test to execute the VSweep user module.
12. Select **Configure**.
13. In the right pane Test Settings tab, from the User Libraries list, select `my_2nd_lib`.
14. From the User Modules list, select the `Vsweep` user module. A default schematic and group of parameters are displayed for the UTM.
15. For Vstart, enter the sweep values.
16. Select **Run**.
17. Select **Analyze**.

At the conclusion of execution, review the results in the Analyze sheet. If you connected a 1 kΩ resistor between SMU1 and GNDU, used the default UTM parameter values, and executed the UTM successfully, the results should be similar to the results in the following figure. The current/voltage ratio for each row of results should be approximately 1 mA / V.

In the example in the following figure, a code of `0` is returned. This means that the user module executed with no errors.

Figure 21: Checking the VSweep user module



	A	B	C
1	VSweep	Imeas	Vforce
2	0	-623.4790E-9	000.0000E-3
3		501.7580E-6	500.0000E-3
4		1.0036E-3	1.0000E+0
5		1.5028E-3	1.5000E+0
6		2.0048E-3	2.0000E+0
7		2.5040E-3	2.5000E+0
8		3.0054E-3	3.0000E+0
9		3.5065E-3	3.5000E+0
10		4.0076E-3	4.0000E+0
11		4.5071E-3	4.5000E+0
12		5.0077E-3	5.0000E+0

Tutorial: Creating a user module that returns data arrays in real time

This tutorial helps you use array variables in KULT and return real-time data. It also illustrates the use of return types (or codes), and the use of two functions from the Keithley Linear Parametric Test Library (LPTLib).

NOTE

The steps that were detailed in Tutorial: Creating a new user library and user module (on page 2-3) are abbreviated in this tutorial.

Naming new user library and new VSweepRT user module

To name new user library and new VSweep user module:

1. Start **KULT**.
2. Select **File > New Library**.
3. In the Enter Library dialog that appears, enter `my_2nd_lib` as the new user library name.
4. Select **OK**.
5. Select **File > New Module**.
6. In the **Module** text box at the top of the KULT dialog, enter `VSweepRT` as the new module name.
7. Select **Apply**.

Entering the VSweepRT user-module return type

Select **int** from the Return Type list. This configures the VSweepRT user module to generate an integer return value.

Entering the VSweepRT user-module code

In the module code-entry area, enter the C code below for the VSweep user module. To simplify code entry, open the KULT dialog to full screen view.

```
/* VSweep module
   --------------
Sweeps through specified V range & measures I, using specified number of points.
Places forced voltage & measured current values (Vforce and Imeas) in output
   arrays.

*/
double vstep, v;   /* Declaration of module internal variables. */
int i;
if ( (Vstart == Vstop) )   /* Stops execution and returns -1 if */
    return( -1 );   /* sweep range is zero. */

vstep = (Vstop-Vstart) / (NumVPoints -1);   /* Calculates V-increment size. */

for(i=0, v = Vstart; i < NumVPoints; i++)   /* Loops through specified number of */
   /* points. */
{
forcev(SMU1, v);   /* LPTLib function forceX, which forces a V or I.  */
measi(SMU1, Imeas);   /* LPTLib function measX, which measures a V or I. */
PostDataDouble("Vforce", v); /* Returns Vforce for display in UTM Sheet. */
v = v + vstep;   /* Increments the forced voltage. */
 }
return( 0 ); /* Returns zero if execution is OK. */
```

Entering the VSweepRT user-module parameters

This example uses the double-precision `D_ARRAY_T` array type. The `D_ARRAY_T`, `I_ARRAY_T`, and `F_ARRAY_T` are special array types that are unique to KULT. For each of these array types, you cannot enter values in the Default, Min, and Max fields.

NOTE

When executing the Vsweep user module in a UTM, the start and stop voltages (Vstart and Vstop) must differ. Otherwise, the first return statement in the code halts execution and returns an error number (−1). When a user module is executed using a Clarius UTM, this return code is stored in the UTM Data worksheet. The return code is stored in a column that is labeled with the user-module name.

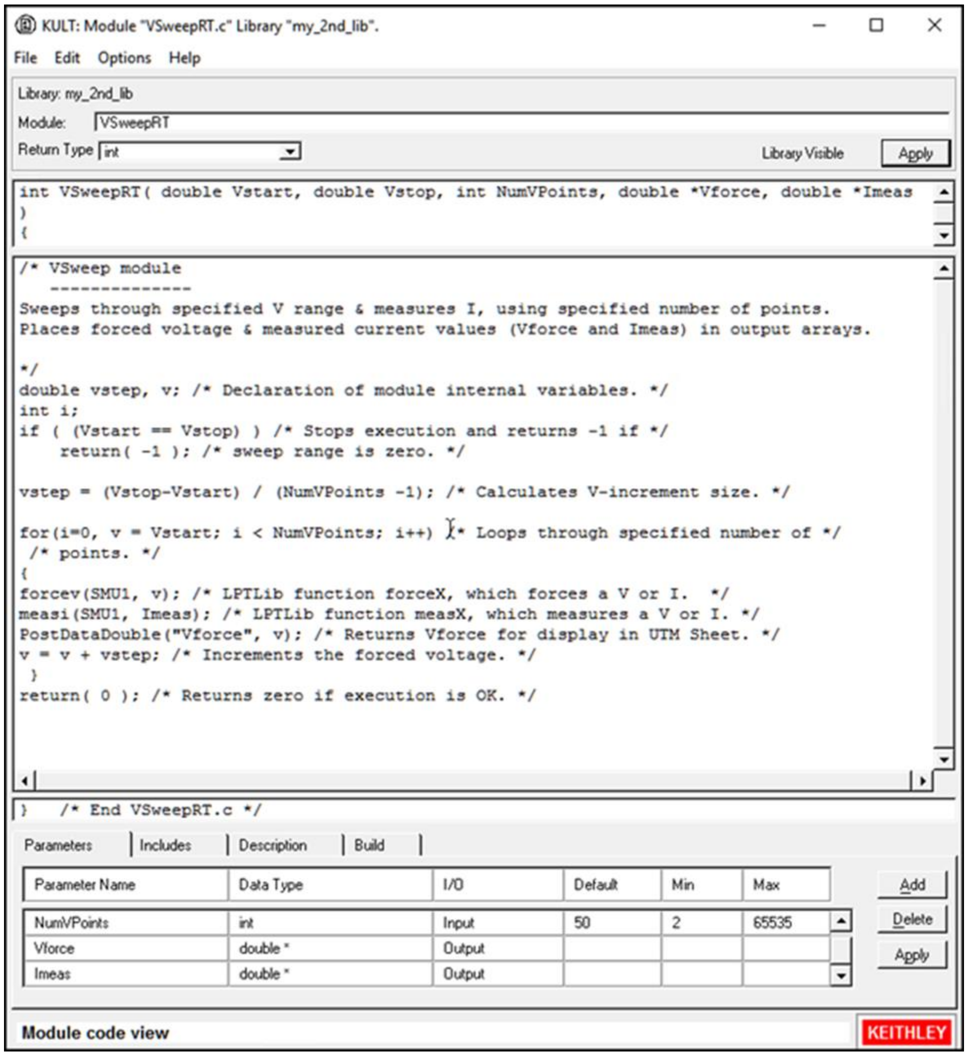
To enter the parameters for the code:

1. Select the **Parameters** tab.
2. Enter the information for the two voltage input parameters, as shown in the following table. Select the **Add** button before adding each new parameter.

Parameter name	Data type	I/O	Default	Min	Max
Vstart	double	Input	0	−200	200
Vstop	double	Input	5	−200	200
NumVPoints	int	Input	50	2	65535
Vforce	double *	Output	—	—	—
Imeas	double *	Output	—	—	—

3. Select **Apply**. In the Parameters tab, the function prototype now includes the declared parameters, as shown in the following figure.

Figure 22: VSweepRT user-module dialog after entering and applying code and parameters



Library: my_2nd_lib
Module: VSweepRT
Return Type: int

```
int VSweepRT( double Vstart, double Vstop, int NumVPoints, double *Vforce, double *Imeas
)
{
```

```
/* VSweep module
   --------------
Sweeps through specified V range & measures I, using specified number of points.
Places forced voltage & measured current values (Vforce and Imeas) in output arrays.

*/
double vstep, v; /* Declaration of module internal variables. */
int i;
if ( (Vstart == Vstop) ) /* Stops execution and returns -1 if */
    return( -1 ); /* sweep range is zero. */

vstep = (Vstop-Vstart) / (NumVPoints -1); /* Calculates V-increment size. */

for(i=0, v = Vstart; i < NumVPoints; i++) /* Loops through specified number of */
 /* points. */
{
forcev(SMU1, v); /* LPTLib function forceX, which forces a V or I.  */
measi(SMU1, Imeas); /* LPTLib function measX, which measures a V or I. */
PostDataDouble("Vforce", v); /* Returns Vforce for display in UTM Sheet. */
v = v + vstep; /* Increments the forced voltage. */
 }
return( 0 ); /* Returns zero if execution is OK. */
```

```
}   /* End VSweepRT.c */
```

Parameter Name	Data Type	I/O	Default	Min	Max
NumVPoints	int	Input	50	2	65535
Vforce	double *	Output			
Imeas	double *	Output			

Entering the VSweepRT user-module header files

You do not need to enter any header files for the VSweepRT user module. The default `keithley.h` header file is sufficient.

Documenting the VSweepRT user module

Select the Description tab and enter documentation for the user module, based on the comments provided in the code and other information about the module.

Saving the VSweepRT user module

From the **File** menu, select **Save Module**.

Building the VSweepRT user module

To build the user module:

1. Select the **Build** tab at the bottom of the dialog to open the Build tab area.
2. In the **Options** menu, select **Build Library**. You should not see error messages.

NOTE

If you do see error messages, check for typographic errors, then fix and rebuild the user module. If necessary, review Finding build errors (on page 2-11).

Checking the VSweepRT user module

Check the user module by creating and executing a UTM in Clarius.

To check the user module:

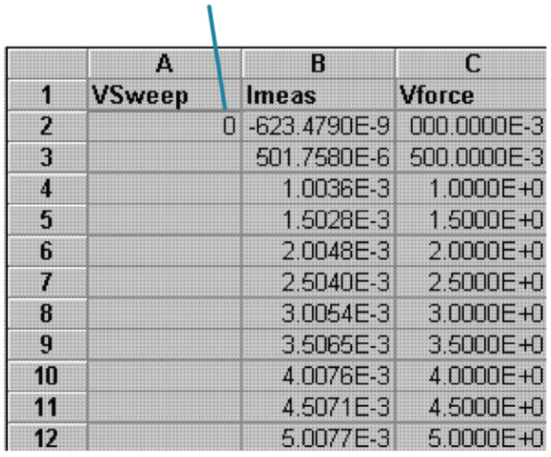
1. Connect a 1 kΩ resistor between the FORCE terminal of the ground unit (GNDU) and the FORCE terminal of SMU1.
2. Instead of creating a new project, reuse the `UserModCheck` project that you created in Tutorial: Creating a new user library and user module (on page 2-3).
3. Choose **Select**.
4. Select the **Devices** tab.
5. Select the 2-wire-resistor.
6. Choose **Select**.
7. Select the **Tests** tab.

8. For the Custom Test, select **Choose a test from the pre-programmed library (UTM)**.
9. Drag **Custom Test** to the project tree. The test has a red triangle next to it to indicate that it is not configured.
10. Select **Rename**.
11. Enter the name `v_sweepRT_chk`. You will use this UTM test to execute the VSweepRT user module.
12. Select **Configure**.
13. In the right pane Test Settings tab, from the User Libraries list, select `my_2nd_lib`.
14. From the User Modules list, select the `VsweepRT` user module. A default schematic and group of parameters are displayed for the UTM.
15. For Vstart, enter the sweep values.
16. Select **Run**.
17. Select **Analyze**.

At the conclusion of execution, review the results in the Analyze sheet. If you connected a 1 kΩ resistor between SMU1 and GNDU, used the default UTM parameter values, and executed the UTM successfully, the results should be similar to the results in the following figure. The current/voltage ratio for each row of results should be approximately 1 mA / V.

In the example in the following figure, a code of `0` is returned. This means that the user module executed with no errors.

Figure 23: Checking the VSweep user module



	A	B	C
1	VSweep	Imeas	Vforce
2	0	-623.4790E-9	000.0000E-3
3		501.7580E-6	500.0000E-3
4		1.0036E-3	1.0000E+0
5		1.5028E-3	1.5000E+0
6		2.0048E-3	2.0000E+0
7		2.5040E-3	2.5000E+0
8		3.0054E-3	3.0000E+0
9		3.5065E-3	3.5000E+0
10		4.0076E-3	4.0000E+0
11		4.5071E-3	4.5000E+0
12		5.0077E-3	5.0000E+0

Tutorial: Calling one user module from within another

KULT allows a user module to call other user modules. A called user module may be in the same user library as the calling module or may be in another user library. This section provides a brief tutorial that illustrates application of such dependencies. It also illustrates the **File > Copy Module** command.

In this tutorial, you create a new user module using two user modules that were created in the previous tutorials: Creating a new user library and user module (on page 2-3) and Creating a user module that returns data arrays (on page 2-14):

- The VSweep user module in the `my_2nd_lib` user library, a copy of which is used as the dependent user library.
- The `TwoTonesTwice` user module, in the `my_1st_lib user` library, which is the independent user library that will be called by the VSweep user module.

A copy of the VSweep user module, called `VSweepBeep`, calls the `TwoTonesTwice` user module to signal the end of execution.

Creating the VSweepBeep user module by copying an existing user module

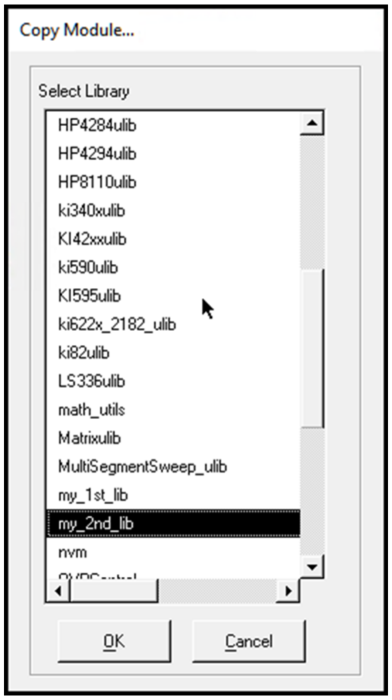
Open the Vsweep user module:

1. Start **KULT**.
2. Select **File > Open Library**.
3. Select `my_2nd_lib` from the list.
4. Select **OK**.
5. Select **File > Open Module**.
6. Select `VSweep.c` from the list.
7. Select **OK**.

Copy `VSweep.c` to the new user module `VSweepBeep`:

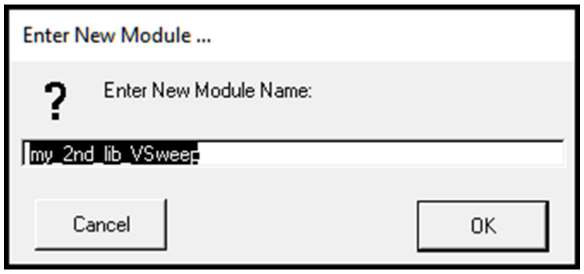
1. Select **File > Copy Module**. The Copy Module list shown in the following figure opens.

Figure 24: Copy Module list



2. Select `my_2nd_lib` (in this case, the user library for the copy is the same as the user library for the source).
3. Select **OK**. The Enter New Module dialog opens, as shown here.

Figure 25: Enter New Module dialog



4. Enter the name **VSweepBeep**.
5. Select **OK**.

NOTE

The name of the user module must not duplicate the name of any existing user module or user library in the entire collection of user libraries.

More than one collection of user libraries can be maintained and accessed, each collection residing in a separate `usrlib`. However, only one `usrlib` can be active at a time. For more information, refer to the Managing user libraries (on page 3-1).

KULT creates a copy of the user module under the new name and displays a message indicating the need to rebuild the user library. You can skip the rebuild for now. Continue with the next step.

Open the new `VSweepBeep` user module:

1. Select **File > Open Module.**
2. Select **VSweepBeep.c** from the list. The KULT dialog displays the `VSweepBeep` user module.

NOTE

You can also create a copy of the presently open user module in the same user library as follows:

1. Enter a new name in the User Module text box.
2. Select **Apply**. Before using the user module, you must save and rebuild the user library.

Calling independent user module from VSweepBeep user module

To call the `TwoTonesTwice` user module at the end of the `VSweepBeep` user module:

1. At the end of `VSweepBeep`, immediately before the `return(0)` statement, add the following statement:

```
TwoTonesTwice(Freq1, Freq2); /* Beeps 4X at end of sweep. */
```

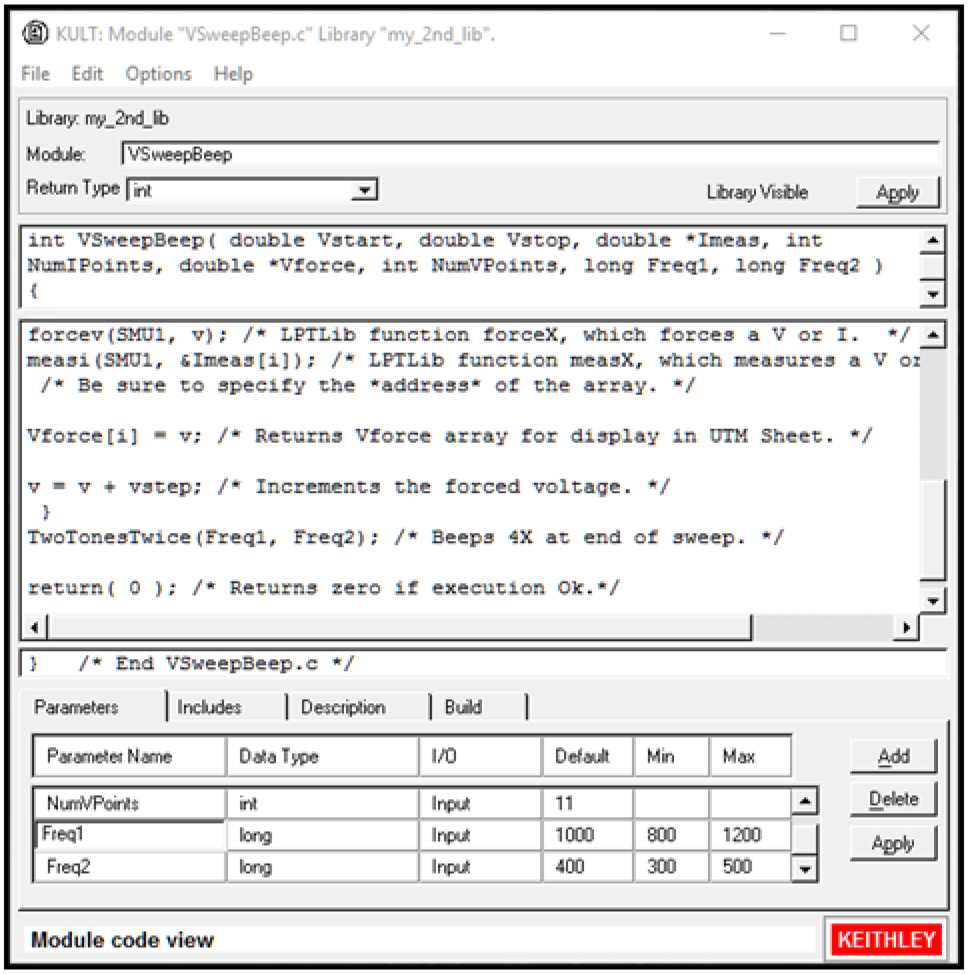
2. In the Parameters tab area, add the **Freq1** and **Freq2** parameters with the values shown in the following table, as you did when you created the `TwoTonesTwice` user module, changing the Default, Min, and Max values as needed.

Parameter entries for the called user module, TwoTonesTwice

Parameter name	Data type	I/O	Default	Min	Max
Freq1	long	Input	1000	800	1200
Freq2	long	Input	400	300	500

3. Select **Apply**. The **Freq1** and **Freq2** parameters are added to the function prototype as shown in the following figure.

Figure 26: Completed VSweepBeep user module



Library: my_2nd_lib
Module: VSweepBeep
Return Type int

```
int VSweepBeep( double Vstart, double Vstop, double *Imeas, int
NumIPoints, double *Vforce, int NumVPoints, long Freq1, long Freq2 )
{
```

```
forcev(SMU1, v); /* LPTLib function forceX, which forces a V or I.  */
measi(SMU1, &Imeas[i]); /* LPTLib function measX, which measures a V or
 /* Be sure to specify the *address* of the array. */

Vforce[i] = v; /* Returns Vforce array for display in UTM Sheet. */

v = v + vstep; /* Increments the forced voltage. */
 }
TwoTonesTwice(Freq1, Freq2); /* Beeps 4X at end of sweep. */

return( 0 ); /* Returns zero if execution Ok.*/
```

```
}   /* End VSweepBeep.c */
```

Parameter Name	Data Type	I/O	Default	Min	Max
NumVPoints	int	Input	11		
Freq1	long	Input	1000	800	1200
Freq2	long	Input	400	300	500

Specifying user library dependencies in VSweepBeep user module

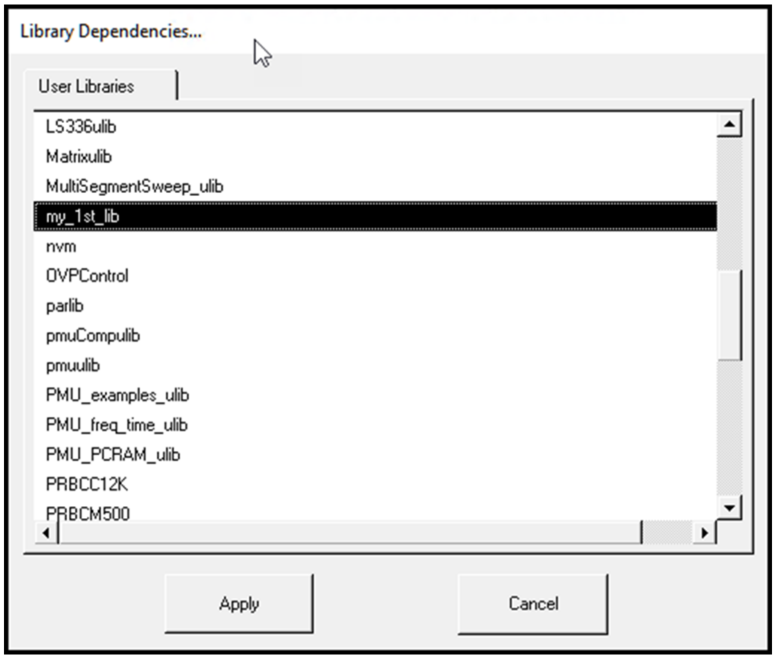
Before building the presently open user module, you must specify all user libraries on which the user module depends (the other user libraries that contain user modules that are called).

The `VSweepBeep` user module depends on the `my_1st_lib` user library.

To specify this dependency:

1. In the **Options** menu, select **Library Dependencies**. The Library Dependencies list opens, as shown here.

Figure 27: Library Dependencies list



LS336ulib
Matrixulib
MultiSegmentSweep_ulib
my_1st_lib
nvm
OVPControl
parlib
pmuCompulib
pmuulib
PMU_examples_ulib
PMU_freq_time_ulib
PMU_PCRAM_ulib
PRBCC12K
PRBCM500

In general, in the Library Dependencies list box, select all user libraries on which the presently open user module depends (each selection toggles on and off). For the `VSweepBeep` module, select `my_1st_lib`.

2. Select **Apply**.

Building the VSweep user module

To build the `VSweepBeep` user module:

1. Save the `VSweepBeep` user module.
2. Select the **Build** tab at the bottom of the dialog to open the Build tab area.
3. In the **Options** menu, select **Build Library**. The user library builds. You should not see error messages.

NOTE

If you see error messages, check for typographical errors; then fix and rebuild the module. If necessary, review Finding build errors (on page 2-11).

Checking the VSweepBeep user module

Check the user module as you did in the previous tutorials by creating and executing a user test module (UTM) in Clarius. Refer to Checking the user module (on page 2-12) for details.

This tutorial is almost identical to Tutorial: Creating a user module that returns data arrays (on page 2-14) except that four beeps should sound at the end of execution.

Before proceeding:

1. Connect a 1 kΩ resistor between the FORCE terminal of the GNDU and the FORCE terminal of SMU1.
2. Instead of creating a new project, reuse the `UserModCheck` project that you created in Tutorial: Creating a new user library and user module (on page 2-3). Add to this project a UTM called `v_sweep_bp_chk`.
3. Configure the `v_sweep_bp_chk` UTM to execute the `VSweepBeep` user module, which is found in the `my_2nd_lib` user library.
4. Run the `v_sweep_bp_chk` UTM. Near the end of a successful execution, you should hear a sequence of four tones, sounded at alternating frequencies.
5. At the conclusion of execution, review the results in the Analyze sheet (or the Graph document, if configured). If you connected a 1 kΩ resistor between SMU1 and GNDU, used the default UTM parameter values, and executed the UTM successfully, your results should be similar to the results shown in Checking the VSweep user module (on page 2-18). The current/voltage ratio for each row of results should be approximately 1 mA/V.

Tutorial: Customizing a user test module (UTM)

This tutorial demonstrates how to modify a user module using KULT. In the `ivswitch` project, there is a test named `rdson`. The `rdson` test measures the drain-to-source resistance of a saturated N-channel MOSFET as follows:

1. Applies 2 V to the gate (V_g) to saturate the MOSFET.
2. Applies 3 V to the drain (V_{d1}) and performs a current measurement (I_{d1}).
3. Applies 5 V to the drain (V_{d2}) and performs another current measurement (I_{d2}).

   Calculates the drain-to-source resistance `rdson` as follows:

   `rdson` = $(V_{d2}-V_{d1}) / (I_{d2}-I_{d1})$

The `rdson` test has a potential shortcoming. If the drain current is noisy, the two current measurements may not be representative of the actual drain current. Therefore, the calculated resistance may be incorrect.

In this example, the user module is modified in KULT so that ten current measurements are made at V_{d1} and ten more at V_{d2}. The current readings at V_{d1} are averaged to yield I_{d1}, and the current readings at V_{d2} are averaged to yield I_{d2}. Using averaged current readings smooths out the noise.

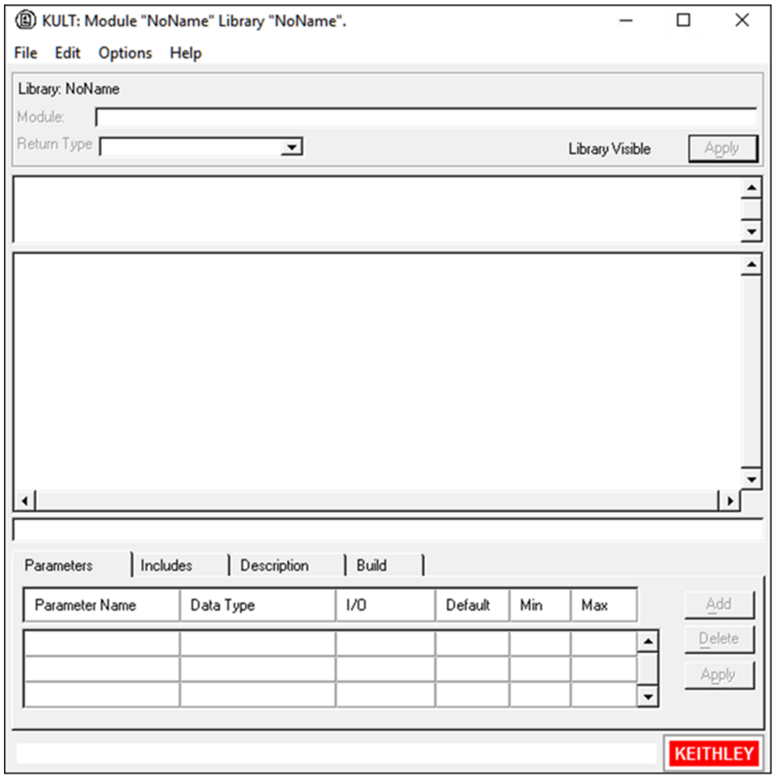
The modified test, `rdsonAvg`, measures the drain-to-source resistance of a saturated MOSFET. The MOSFET is tested as follows when `rdsonAvg` is executed:

1. Applies 2 V to the gate (V_g) to saturate the MOSFET.
2. Applies 3 V to the drain (V_{d1}) and makes ten current measurements.
3. Averages the 10 current readings to yield a single reading (I_{d1}).
4. Applies 5 V to the drain (V_{d2}) and makes ten more current measurements.
5. Averages the ten current readings to yield a single reading (I_{d2}).
6. Calculates the drain-to-source resistance (`rdsonAvg`) as follows:
   `rdsonAvg` = (V_{d2}-V_{d1}) / (I_{d2}-I_{d1})

Open KULT

From the desktop, open the KULT tool by double-clicking the KULT icon. The KULT main dialog is shown in the following figure.

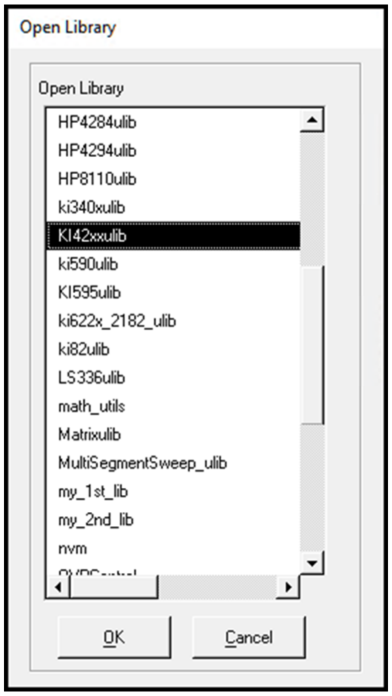
Figure 28: KULT main dialog



Open the KI42xxulib user library

1. Select **File > Open Library**.
2. From the Open Library dialog, select **KI42xxulib**.

Figure 29: KULT Open Library dialog

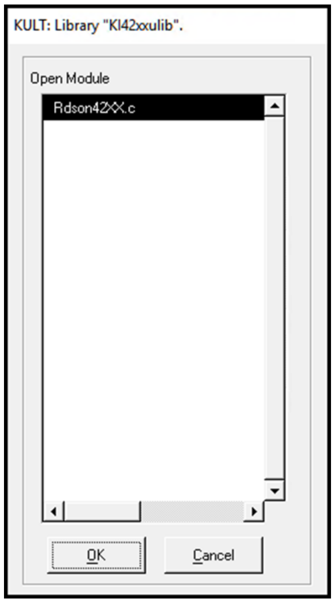


3. Select **OK**.

Open the Rdson42XX user module

1. From the **File** menu, select **Open Module**.
2. From the Open Module dialog, select **Rdson42XX.c**, as shown in the following figure.

Figure 30: KULT Open Module dialog



3. Select **OK**. The Rdson42XX module opens.

Copy Rdson42XX to RdsonAvg

You create the new module by copying the `Rdson42XX` module to a module named `RdsonAvg` and then making the appropriate changes to the test module.

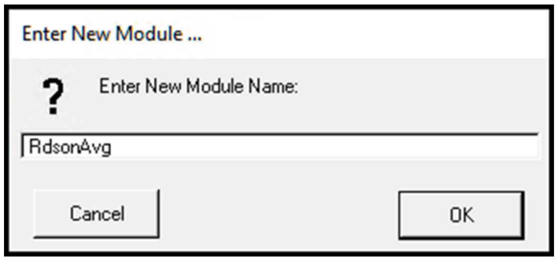
NOTE

When naming a user module, conform to case-sensitive C programming language naming conventions. Do not duplicate names of existing user modules or user libraries.

To create the new module:

1. From the **File** menu, select **Copy Module**.
2. Select the library for the module. From the Copy Module dialog, select **KI42xxulib**.
3. Select **OK**.
4. In the Enter New Module dialog, type in `RdsonAvg`.

Figure 31: Enter New Module Name dialog



5. Select **OK**. A reminder that the library using the new module needs to be built is displayed.
6. Select **OK**.

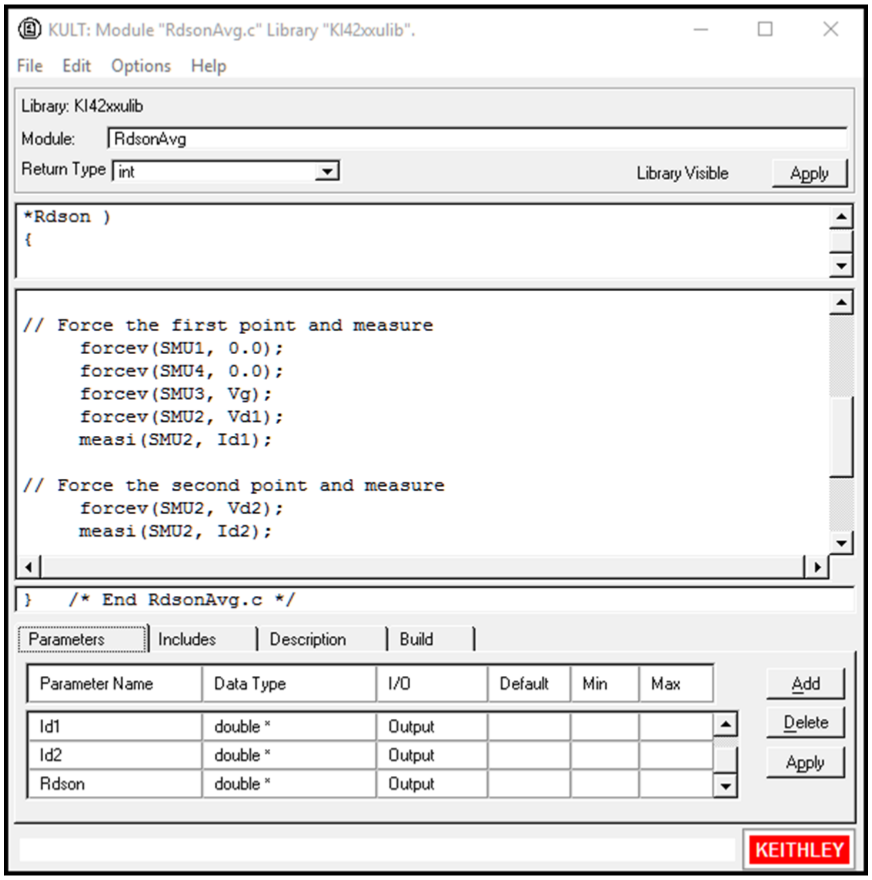
Open and modify the RdsonAvg user module

To open the user module:

1. From the **File** menu, select **Open Module**.
2. Select **RdsonAvg.c** from the Open Module dialog.

   The `RdsonAvg` module is shown in the following figure.

Figure 32: KULT module dialog



Library: KI42xxulib

Module: RdsonAvg

Return Type: int

```
*Rdson )
{
```

```
// Force the first point and measure
     forcev(SMU1, 0.0);
     forcev(SMU4, 0.0);
     forcev(SMU3, Vg);
     forcev(SMU2, Vd1);
     measi(SMU2, Id1);

// Force the second point and measure
     forcev(SMU2, Vd2);
     measi(SMU2, Id2);
```

```
}   /* End RdsonAvg.c */
```

Parameter Name	Data Type	I/O	Default	Min	Max
Id1	double *	Output			
Id2	double *	Output			
Rdson	double *	Output			

Modify the user module code

In the user module code, you need to replace the `measi` commands with `avgi` commands. While a `measi` command makes a single measurement, an `avgi` command makes a specified number of measurements, and then calculates the average reading. For example:

```
avgi(SMU2, Id1, 10, 0.01);
```

For the above command, SMU2 makes 10 current measurements and then calculates the average reading (`Id1`). The 0.01 parameter is the delay between measurements (10 ms).

The source code for the module is in the module code area of the dialog. In this area, make the following changes.

Under `Force the first point and measure`, change the line:

```
measi(SMU2, Id1);
```

to

```
avgi(SMU2, Id1, 10, 0.01); // Make averaged I measurement
```

Under `Force the second point and measure`, change the line:

```
measi(SMU2, Id2);
```

to

```
avgi(SMU2, Id2, 10, 0.01); // Make averaged I measurement
```

Change the line:

```
*Rdson = (Vd2-Vd1)/(*Id2- *Id1);   // Calculate Rdson
```

to

```
*RdsonAvg = (Vd2-Vd1)/(*Id2- *Id1);   // Calculate RdsonAvg
```

Change a parameter name

Change the name of the `Rdson` parameter:

1. Select the Parameters tab.
2. Scroll down to the parameter `Rdson`.
3. Select the name and change it to `RdsonAvg`.
4. Select **Apply**.

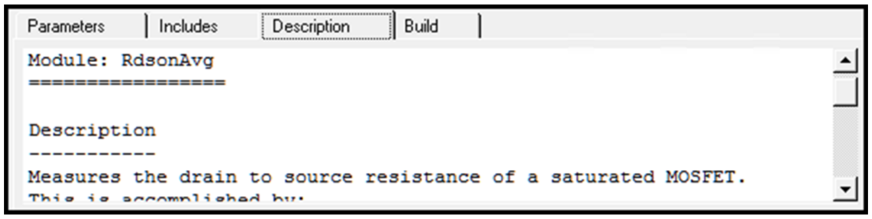
Change the module description

In Clarius, any user test modules (UTMs) that are connected to this user module show the text that is entered on the Description tab in KULT.

To change the module description:

1. Select the **Description** tab.
2. Above `DESCRIPTION`, change `MODULE: Rdson42xx` to `MODULE: RdsonAvg`, as shown in the following figure.
3. Replace all occurrences of `Rdson` with `RdsonAvg`.

Figure 33: User module description



```
Module: RdsonAvg
=================

Description
-----------
Measures the drain to source resistance of a saturated MOSFET.
```

Save and build the modified library

You must save and also rebuild the library to ensure that the new module is available for use by Clarius user test modules (UTMs).

To save and build the user module and library:

1. Select **File > Save Module**.
2. Select **Options > Build Library**. A dialog is displayed that indicates the build is in process.

Add the new UTM to the ivswitch project

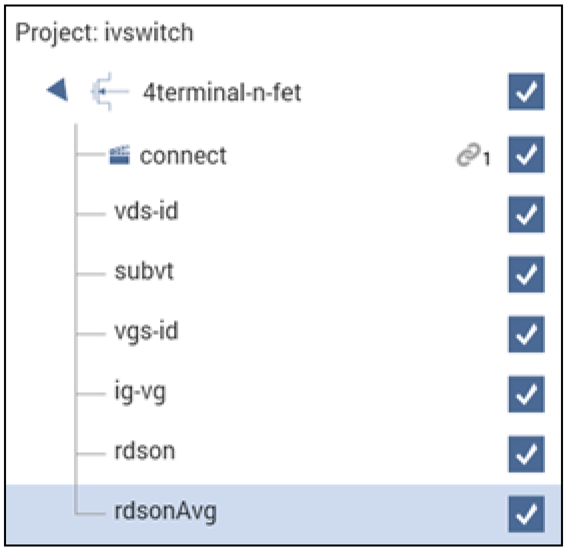
To add rdsonAvg to the ivswitch project:

1. Choose **Select**.
2. Select **Projects**.
3. In the Search box, enter **ivswitch** and select **Search**. The Library displays the I-V Switch Project (ivswitch).
4. Select **Create**. The `ivswitch` project replaces the previous project in the project tree.
5. Select the **Tests** tab.

6. For the Custom Test, select **Choose a test from the pre-programmed library (UTM)**.
7. Drag **Custom Test** to the project tree. The test has a red triangle next to it to indicate that it is not configured.
8. Select **Rename**.
9. Enter **rdsonAvg** and press **Enter**.
10. In the project tree, drag **rdsonAvg** to the `4terminal-n-fet` device, after the `rdson` test.
11. Choose **Configure**.
12. In the Test Settings pane, from the User Libraries list, select **KI42xxulib**.
13. From the User Modules list, select **Rdson42XX**.
14. Select **Save**.

The project tree for the `ivswitch` project with `rdsonAvg` added is shown in the following figure.

Figure 34: Project tree with rdsonAvg added to 4terminal-n-fet device



Tutorial: Creating a user module for stepping or sweeping

This section provides a tutorial that helps you set up a user test module (UTM) that supports stepping or sweeping. This example is similar to the `vds-id` test. For each gate voltage step, the test sweeps the drain voltage.

NOTE

Most of the basic steps that were detailed in Tutorial: Creating a new user library and user module (on page 2-3) are abbreviated in this tutorial. This tutorial adds a user module to the library `my_2nd_lib`, which was created in Tutorial: Creating a user module that returns data arrays (on page 2-14).

Name a new user module

To name new user library and new user module:

1. Start KULT.
2. Select **File > Open Library**.
3. Select **my_2nd_lib**.
4. Select **OK**.
5. Select **File > New Module**.
6. For Module, enter **vds_id_step_sweep**.
7. Select **Apply**.

Entering the return type

From the Return Type list, select **int**. This configures the user module to generate an integer return value.

Entering the user-module code

In the module code-entry area, enter the C language code below for the user module. To simplify code entry, open the KULT dialog to full-screen view.

```
int retCode = 0;  // This module returns an error or success code to Clarius (shown
   in the first column of the data grid).
int stepSteps = 1;
int sweepSteps = 8;
int i = 0;
int j = 0;
int stepperID = 1;
double vg = VgStart;
double stepSimTime = 5000.0;  // The time to simulate acquisition of one step data.
double pointDelay = 1.0;      // Simulated delay between single data points.
double vd = VdStart;
double id = vd / 1e6; // Simulate id current.
double vgScale = 1.0; // Simulate shift in id data between different steps.
char vgName[32];  // Output names for PostDataDouble data transfer to Clarius.
char vdName[32];
char idName[32];

if (VdStep == 0.0 || VgStep == 0.0)
{
return -1;  // Invalid input parameters
}

stepSteps = fabs((VgStop - VgStart) / VgStep) + 1;
sweepSteps = fabs((VdStop - VdStart) / VdStep) + 1;
pointDelay = stepSimTime / sweepSteps;

for (i = 0; i < stepSteps; i++)
{
vd = VdStart;
id = vd / 1e6;
// Define output column names for each step (must include stepperID).
stepperID = i + 1;
sprintf(vgName, "OutVg(%d)", stepperID);
sprintf(vdName, "OutVd(%d)", stepperID);
sprintf(idName, "OutId(%d)", stepperID);

for (j = 0; j < sweepSteps; j++)
{
PostDataDouble(vgName, vg);
PostDataDouble(vdName, vd);
PostDataDouble(idName, id);
Sleep(pointDelay);

vd += VdStep;
id = sqrt(vd * vgScale) / 1e6;
}
vg += VgStep;
vgScale += 0.2;
}

return retCode;
```

Entering the user-module parameters

NOTE

When the user module is executed in a UTM, the start and stop voltages must differ. Otherwise, the first return statement in the code halts execution and returns an error number (−1). This return code is stored in the Analyze sheet for the test, in a column that is labeled with the user-module name.

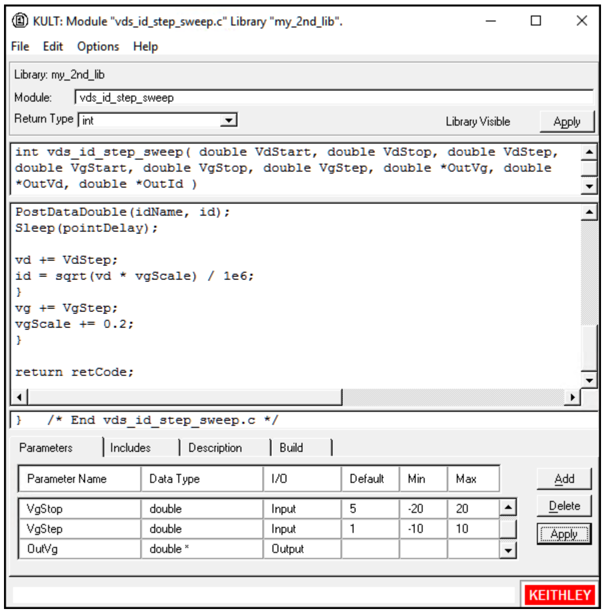
To enter the parameters for the code:

1. Select the **Parameters** tab.
2. Enter the information for the two voltage input parameters, as shown in the following table. Select the **Add** button before adding each new parameter.

Parameter name	Data type	I/O	Default	Min	Max
VdStart	double	Input	0	−2000	2000
VdStop	double	Input	10	−2000	2000
VdStep	double	Input	0.2	−1000	1000
VgStart	double	Input	1	−20	20
VgStop	double	Input	5	−20	20
VgStep	double	Input	1	−10	10
OutVg	double *	Output	—	—	—
OutVd	double *	Output	—	—	—
OutId	double *	Output	—	—	—

3. Select **Apply**. In the Parameters tab, the function prototype now includes the declared parameters, as shown in the following figure.

Figure 35: Parameters for the vds_id_step_sweep user module



Library: my_2nd_lib
Module: vds_id_step_sweep
Return Type: int

```
int vds_id_step_sweep( double VdStart, double VdStop, double VdStep,
double VgStart, double VgStop, double VgStep, double *OutVg, double
*OutVd, double *OutId )
```

```
PostDataDouble(idName, id);
Sleep(pointDelay);

vd += VdStep;
id = sqrt(vd * vgScale) / 1e6;
}
vg += VgStep;
vgScale += 0.2;
}

return retCode;
```

```
}   /* End vds_id_step_sweep.c */
```

Parameter Name	Data Type	I/O	Default	Min	Max
VgStop	double	Input	5	-20	20
VgStep	double	Input	1	-10	10
OutVg	double *	Output			

Enter the user-module header files

You do not need to enter any header files for the vds_id_step_sweep user module. The default `keithley.h` header file is sufficient.

Documenting the user module

Select the **Description** tab and enter documentation for the user module, based on the comments provided in the code and other information about the module.

Saving the user module

From the **File** menu, select **Save Module**.

Building the user module

To build the user module:

1. Select the **Build** tab at the bottom of the dialog to open the Build tab area.
2. In the **Options** menu, select **Build Library**.
3. Scroll down in the KULT Build Library dialog. You should not see error messages.
4. Select **OK**.

NOTE

If you do see error messages, check for typographical errors, then fix and rebuild the user module. If necessary, review Finding build errors (on page 2-11).

Setting up the user interface of the user module

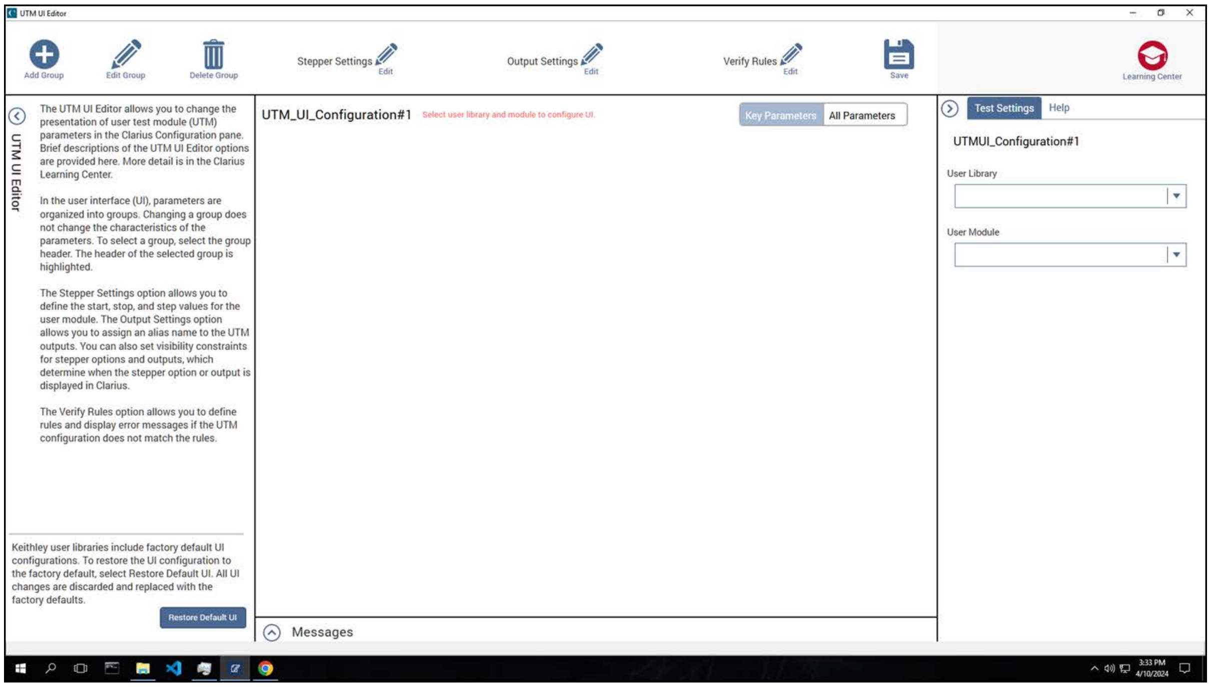
On the Clarius Configure pane, the default user interface for a user module shows an image of the test device and all parameters in one group. You can change the image and how the parameters are grouped using the UTM UI Editor tool.

This example briefly describes how to use the UTM UI Editor tool. For more detail, refer to “Define the UTM user interface” in the *Model 4200A-SCS Clarius User’s Manual*.

To set up the user interface:

1. Close Clarius.
2. From the Windows Start menu, select **Keithley Instruments > UTM UI Editor**. The UTM UI Editor application opens, as shown in the following figure.

Figure 36: UTM UI Editor application



The UTM UI Editor allows you to change the presentation of user test module (UTM) parameters in the Clarius Configuration pane. Brief descriptions of the UTM UI Editor options are provided here. More detail is in the Clarius Learning Center.

In the user interface (UI), parameters are organized into groups. Changing a group does not change the characteristics of the parameters. To select a group, select the group header. The header of the selected group is highlighted.

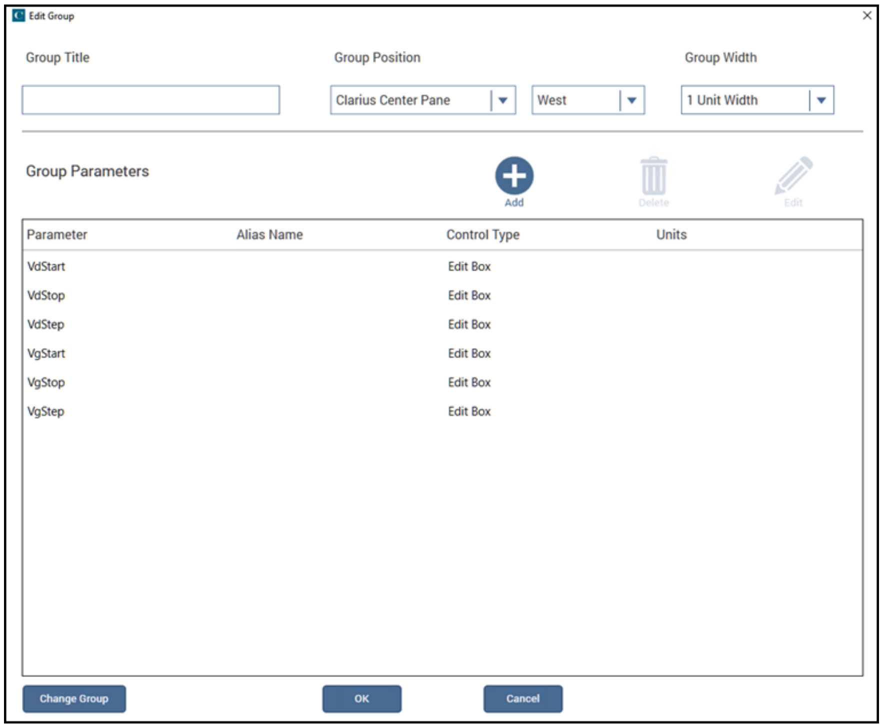
The Stepper Settings option allows you to define the start, stop, and step values for the user module. The Output Settings option allows you to assign an alias name to the UTM outputs. You can also set visibility constraints for stepper options and outputs, which determine when the stepper option or output is displayed in Clarius.

The Verify Rules option allows you to define rules and display error messages if the UTM configuration does not match the rules.

Keithley user libraries include factory default UI configurations. To restore the UI configuration to the factory default, select Restore Default UI. All UI changes are discarded and replaced with the factory defaults.

3. In the right pane Test Settings tab, from the User Libraries list, select `my_2nd_lib`.
4. From the User Modules list, select the `vds_id_step_sweep` user module. A default schematic and group of parameters are displayed for the UTM.
5. Select the header of a group of parameters.
6. Select **Edit Group**. The Edit Group dialog with the default settings is displayed, as shown in the following figure.

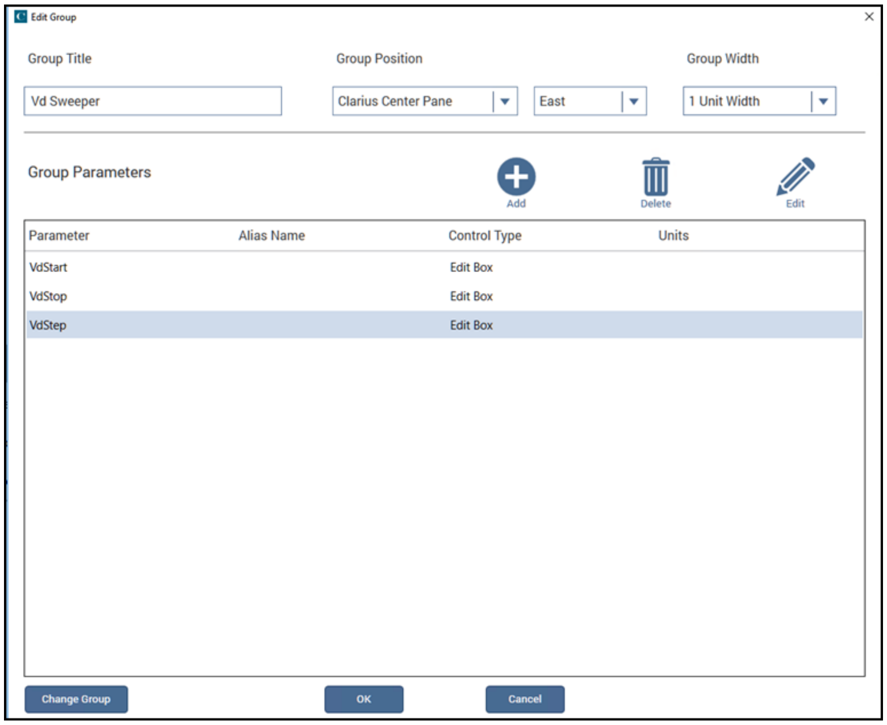
Figure 37: UTM UI Editor with Group 1 selected



Parameter	Alias Name	Control Type	Units
VdStart		Edit Box	
VdStop		Edit Box	
VdStep		Edit Box	
VgStart		Edit Box	
VgStop		Edit Box	
VgStep		Edit Box	

7. In **Group Title**, enter `Vg Stepper`.
8. Set the Group Position to **Clarius Center Pane** and **West**.
9. Delete the **VdStart**, **VdStop**, and **VdStep** parameters from this group.
10. Select **OK** to close the Edit Group dialog.
11. Select **Add Group**.
12. In **Group Title**, enter `Vd Sweeper`.
13. Set the Group Position to **Clarius Center Pane** and **East**.
14. Select **Add** three times. This adds the **VdStart**, **VdStop**, and **VdStep** parameters to this group, as shown in the following figure.

Figure 38: Vd Sweeper group in the UTM UI Editor

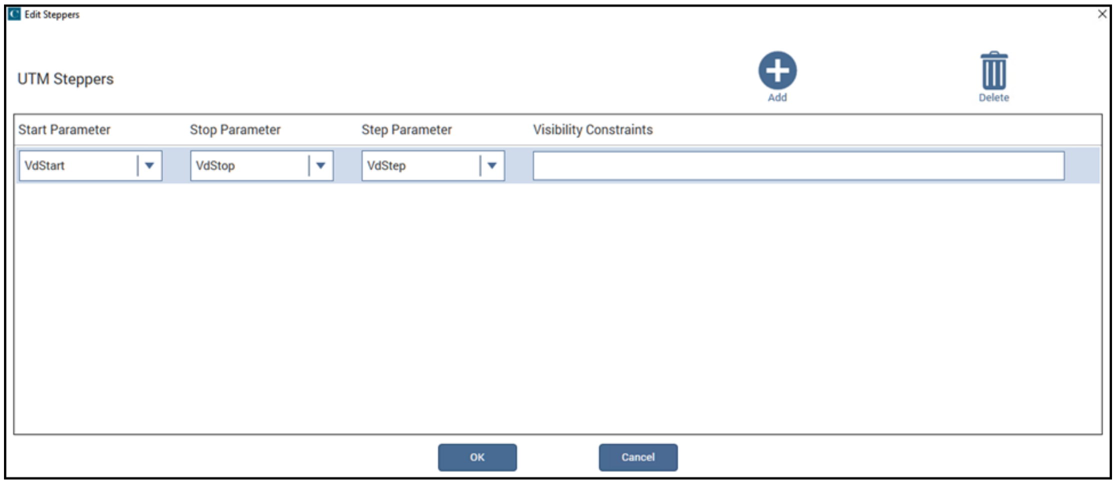


Group Title: Vd Sweeper
Group Position: Clarius Center Pane, East
Group Width: 1 Unit Width

Parameter	Alias Name	Control Type	Units
VdStart		Edit Box	
VdStop		Edit Box	
VdStep		Edit Box	

15. Select **OK**.
16. Select **Stepper Settings**. The Edit Steppers dialog is displayed.
17. Select **Add**. The Vd Sweeper parameters are added.
18. Select **OK**.

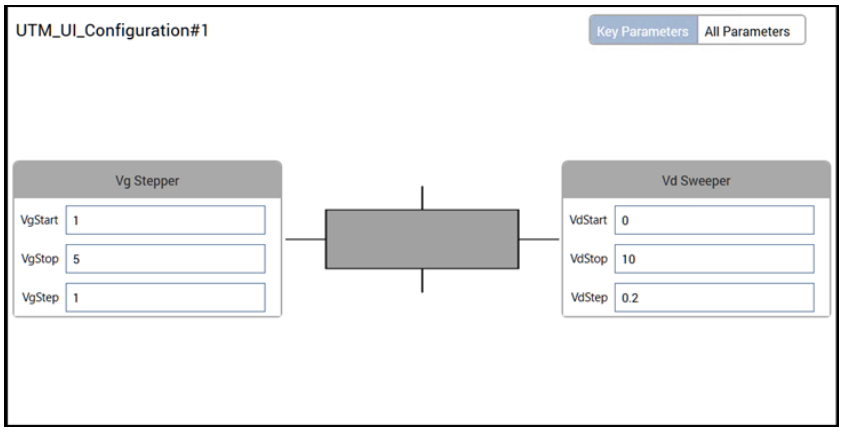
Figure 39: Edit UTM UI for a stepper



Start Parameter	Stop Parameter	Step Parameter	Visibility Constraints
VdStart	VdStop	VdStep	

19. Select **OK**. The new groups are displayed.

Figure 40: vds_id_step_sweep in Configure



UTM_UI_Configuration#1

Vg Stepper

VgStart	1
VgStop	5
VgStep	1

Vd Sweeper

VdStart	0
VdStop	10
VdStep	0.2

Check the user module in Clarius

Check the user module by creating and executing a UTM in Clarius.

To check the user module:

1. Open the `UserModCheck` project that you created in Tutorial: Creating a new user library and user module (on page 2-3).
2. Choose **Select**.
3. Select **Devices**.
4. Select **MOSFET, n-type, t terminal (4terminal-n-fet)**.
5. Choose **Select**.
6. Select the **Tests** tab.
7. For the Custom Test, select **Choose a test from the pre-programmed library (UTM)**.
8. Drag **Custom Test** to the project tree. The test has a red triangle next to it to indicate that it is not configured.
9. Select **Rename**.
10. Enter the name `vds_id_step_sweep`. Use this UTM test to execute the new user module.
11. Select **Configure**.
12. In the right pane Test Settings tab, from the User Libraries list, select `my_2nd_lib`.
13. From the User Module list, select `vds_id_step_sweep`.
14. Select **Run**.
15. Select **Analyze** to review the results. Each output parameter is repeated based on the number of steps.

User module and library management

In this section:

Introduction

Additional features of KULT include:

- Tools to manage user libraries (on page 3-1)
- Dependent user modules and user libraries (on page 3-9)
- Ability to format user module help for the Clarius Help pane (on page 3-14)
- Ability to create project prompts (on page 3-15)

Managing user libraries

This section addresses the following topics:

- Updating and copying user libraries using KULT command-line utilities (on page 3-1) describes two command-line utilities. One utility provides a command-line method to copy user libraries. The other utility provides a means to update user libraries after they are copied.
- Performing other KULT tasks using command-line commands (on page 3-3) describes a series of command-line commands. These commands can be used individually or in a batch file to perform various KULT tasks without opening the KULT user interface.

Updating and copying user libraries using KULT command-line utilities

This section describes the command-line utilities `kultupdate` and `kultcopy`.

Updating user libraries using kultupdate

If you copy user libraries to a new storage location (another user directory or drive), you must use the `kultupdate` utility to update the user libraries. User libraries must be updated to ensure the correctness of all path information, which is built into the library. The `kultupdate` utility rebuilds each user module in the library and also rebuilds the library.

Usage

```
kultupdate <library_name> [options]
```

Options

You can place any of the following options at the `[options]` position in the command:

- `-dep <library_dep_1>...[library_dep_6]`
  Specifies up to six libraries on which `library_name` depends.
- `-hide`
  Hides *`library_name`* so that it is not visible in Clarius.
- `+hide`
  Shows *`library_name`* so that it is visible in Clarius.

Example

Update the `KI590ulib` library in the active user-library directory, which depends on the `Winulib` library:

```
C:\>kultupdate KI590ulib -dep Winulib
```

Copying user libraries using kultcopy

The `kultcopy` utility copies any user library from any accessible storage location to the active user-library directory. The `kultcopy` utility:

- Performs `kultupdate` so that the user library is immediately ready for use. Refer to Updating user libraries using kultupdate (on page 3-2) for more information.
- Copies the user library that is specified by the "Start-In" user-library directory, which is the directory in which you start the `kultcopy` command.

To successfully copy a user library to the active user-library directory, you must start `kultcopy` in the following directory:

```
<source_lib_path>\<source_lib_name>\src
```

This directory is called the "Start-In" directory, where:

- *`<source_lib_path>`* is any accessible user-library directory.
- *`<source_lib_name>`* is the name of the specific user library to be copied.

Usage

```
kultcopy <library_name> [options]
```

Options

Any of the following options may be placed at the `[options]` position in the command:

- `-dep <library_dep_1>...[library_dep_6]`
  Specifies up to six libraries on which *`library_name`* depends.
- `-hide`
  Hides *`library_name`* so that it is not visible in Clarius.
- `+hide`
  Shows *`library_name`* so that it is visible in Clarius.

You can use `kultcopy` restore the original userlib directory. A backup copy of the userlib directory is provided at `c:\s4200\sys\factory\usrlib`.

NOTE

If there are images linked to the original UTMs, the new modules point to the images in the original directory, even though the files for the images were moved. You need to manually change the path to the new directory.

Performing other KULT tasks using command-line commands

The KULT command-line interface lets you load, build, or delete user libraries and add or delete user modules without opening the KULT user interface. This feature is useful when developing and managing user libraries. The commands can be used individually or in a batch file.

The general format for a command line instruction is as follows:

```
kult subcommand -l<library_name> [options] [module]
```

The individual items in the instruction are as follows:

- The item subcommand may be any one of these subcommands:
  - `add_mod`
  - `bld_lib`
  - `del_lib`
  - `del_mod`
  - `gui`
  - `help`

  - `new_lib`
  - `new_mod`
  - `unzip`
  - `zip`
- The item <*`library_name`*> specifies the name of the library involved in the commanded action.
- The item [*`options`*] includes one or more of these options:
  - `-d<`*`directory_name`*`>`
  - `-hide`
  - `+hide`
  - `-dep <library_dep_1>.....[library_dep_6]`
  - `build_type`
- These options are described in the following descriptions of individual subcommands.
- If appropriate to the commanded action, [*`module`*] specifies the name of the involved user module.

The sections that follow describe the subcommands.

gui subcommand

The `gui` subcommand launches the KULT editor.

Usage

`kult gui` *`[option] [type]`*

The -`build_type` option may be placed at the [*`options`*] position in the command. The following *`[type]`* options are available:

- `Release`
  Default option. This option builds the library more efficiently than the `Debug` option.
- `Debug`
  Use this option if you want to use an integrated development environment, such as Visual Studio Code, to debug your source code.

Example

```
kult gui -build_type Release
```

new_lib subcommand

The `new_lib` subcommand lets you create a new user library without any user modules. Its action is equivalent to the following steps in KULT:

- Starting KULT
- Selecting **File > New Library**
- Entering a new library name
- Selecting **OK**
- Selecting **File > Exit**

Usage

```
kult new_lib -l<library_name>
```

The *<library_name>* user library is created in the active user-library directory.

bld_lib subcommand

The `bld_lib` subcommand lets you build a user library from the command line. Its action is equivalent to the following steps in KULT:

- Starting KULT
- Selecting **File > Open Library**
- Selecting the *<library_name>* user library
- Selecting **OK**, selecting **Options > Build Library**
- After the build is completed, selecting **File > Exit**

Usage

```
kult bld_lib -l<library_name> [options]
```

Builds the <*library_name*> user library in the active user-library directory.

Any of the following may be placed at the [*options*] position in the command:

- `-dep <library_dep_1>...[library_dep_6]`
  Specifies up to six user libraries upon which *library_name* depends.

NOTE

Dependent user libraries must be in the active user-library directory. For more information about dependent libraries, refer to Dependent user modules and user libraries (on page 3-9).

- `+hide`
  Hides *`library_name`* so that it is not visible in Clarius.
- `-hide`
  Shows *`library_name`* so that it is visible in Clarius.

NOTE

The `C:\s4200\kiuser\usrlib\<library name>\build` folder is created when you run the `bld_lib` subcommand or select the **Build Library** menu option. This folder can be safely deleted for debugging purposes.

del_lib subcommand

The `del_lib` subcommand lets you delete a library from the command line. Its action is equivalent to the following steps in KULT:

- Starting KULT
- Selecting **File > Delete** Library
- Selecting a user library to be deleted
- Selecting **OK**
- Selecting **File > Exit**

Usage

```
kult del_lib -l<library_name>
```

The `<`*`library_name`*`>` user library is deleted from the active user-library directory.

new_mod subcommand

The `new_mod` subcommand lets you create a new module in a user library. Its action is equivalent to the following steps in KULT:

- Starting KULT
- Selecting **File > Open Library > <library_name>**
- Select **OK**
- **Selecting File > New Module**
- **Enterin**g a new module name
- Selecting **Apply**
- Selecting **File > Exit**

Usage

```
kult new_mod -l<library_name> <module>
```

The *<module>* module is created in the *<library_name>* library.

Where:

- *<library_name>* is the target library into which *<module>* is to be created. It must be in the active user-library directory.
- *<module>* is the new module name.

add_mod subcommand

The `add_mod` subcommand lets you add or copy a user module from one user library (source) to another library (target). Its action is equivalent to the following KULT steps:

- Starting KULT
- Selecting **File > Open Library**
- Selecting the <*source_lib_name*> source library
- Selecting **File > Open Module**
- Selecting the <*module*> source module
- Selecting **File > Copy Module**
- Selecting the <*library_name*> target library
- Entering a target-module name
- Selecting **File > Exit**

NOTE

All user modules must be named uniquely, even if they are duplicates that reside in different user libraries. The `add_mod` subcommand automatically assigns a target-module name that is a derivative of the source-module name. The naming convention is as follows: <*source_library_name*>_<*module*>.

Usage

```
kult add_mod -l<library_name> [-d<source_lib_path>\source_lib_name>\src] <module>
```

Where:

- *<library_name>* is the target library into which *<module>* is to be copied. It must be in the active user-library directory.
- *<source_lib_path>* is any accessible user-library directory.

- `<source_lib_name>` is the name of the specific user library from which `<module>` is to be copied.
- `<module>` is the source user module.

You must use the `-d` option when you execute `add_mod` in a directory other than `<source_lib_path>\<source_lib_name>`.

del_mod subcommand

The `del_mod` subcommand lets you delete a module from the command line. Its action is equivalent to the following steps in KULT:

- Starting KULT
- Selecting **File > Delete Module**
- Selecting a user module to be deleted
- Selecting **OK**
- Selecting **File > Exit**

Usage

```
kult del_mod -l<library_name> <module>
```

Where:

- `<library_name>` is the target library from where `<module>` will be deleted. It must be in the active user-library directory.
- `<module>` is the name of the module to be deleted.

zip subcommand

The *zip* subcommand creates a `.zip` file for a user library.

Usage

```
kult zip -l<library_name> [password] <zipfile_name>
```

The `<library_name>` user library is created in the active user-library directory.

The `[password]` parameter is optional.

unzip subcommand

The *`unzip`* subcommand unzips a file containing a KULT library.

Usage

```
kult unzip [-dest_path] [password] <zipfile_name>
```

Where:

- `[-dest_path]` is the target directory where the file will be unzipped.
- `[password]` is required if the file was compressed using the password parameter in the `zip` subcommand.

The *`<zipfile_name>`* archive is unzipped in the active user-library directory unless the `[-dest_path]` parameter is specified. The `[-dest_path]` parameter should not be used when you import a user library.

help subcommand

The `help` subcommand displays all usage information for subcommands and options.

Usage

```
kult help
```

Dependent user modules and user libraries

KULT allows a user module to call other user modules. A called user module can be in the same user library as the calling module or can be in another user library. When the module that you are creating calls a module in another user library, you must:

1. Select **Options > Library Dependencies**.
2. Specify each called library from the list that is displayed.

You must select user module and user-library dependencies carefully. Observe the following:

- Try to put user modules with interdependencies in the same user library and minimize the interdependencies between libraries. This practice helps to avoid problematic user library dependency loops (`Lib1` relies on `Lib2`, `Lib2` relies on `Lib3`, `Lib3` relies on `Lib1`).
- If a user module in one user library must depend on user modules in other user libraries, take care when selecting the user libraries to be linked with the user module under development. The next section provides guidance.

NOTE

The user libraries to be linked are saved so that future rebuilds do not require the dependencies to be selected again. This information is stored in the `<`*`library_name`*`>_modules.mak` file in the `%KI_KULT_PATH%\<`*`library_name`*`>\kitt_obj` directory.

- Structure dependencies hierarchically to avoid circular dependencies, and then build the dependent user libraries in the correct order. The next two sections provide guidance.

Structuring dependencies hierarchically

You can avoid user library circular dependency by calling user libraries in a hierarchical design, as illustrated in "Hierarchical design for user-library dependencies" below.

Observe the following:

- Design lower-level user modules in the calling hierarchy so that they do not require support from higher-level modules. That is, lower-level user modules should not require calls to higher-level modules to perform their required tasks.
- Use several general-purpose low-level-library user modules to do a task rather than a single, do-all, higher-level-library user module.

You may find it helpful to prefix user modules with the user-library name as an identifier, for example, `liba_ModuleName` for user modules in `liba`. This avoids duplicate user module names and prevents confusion with similarly named modules that are in other user libraries and source files. When you execute the **File > Copy Library** command, KULT automatically appends the user library name to each user module in the new user library name. KULT also appends the library name, as a suggestion, when you execute the **File > Copy Module** command.

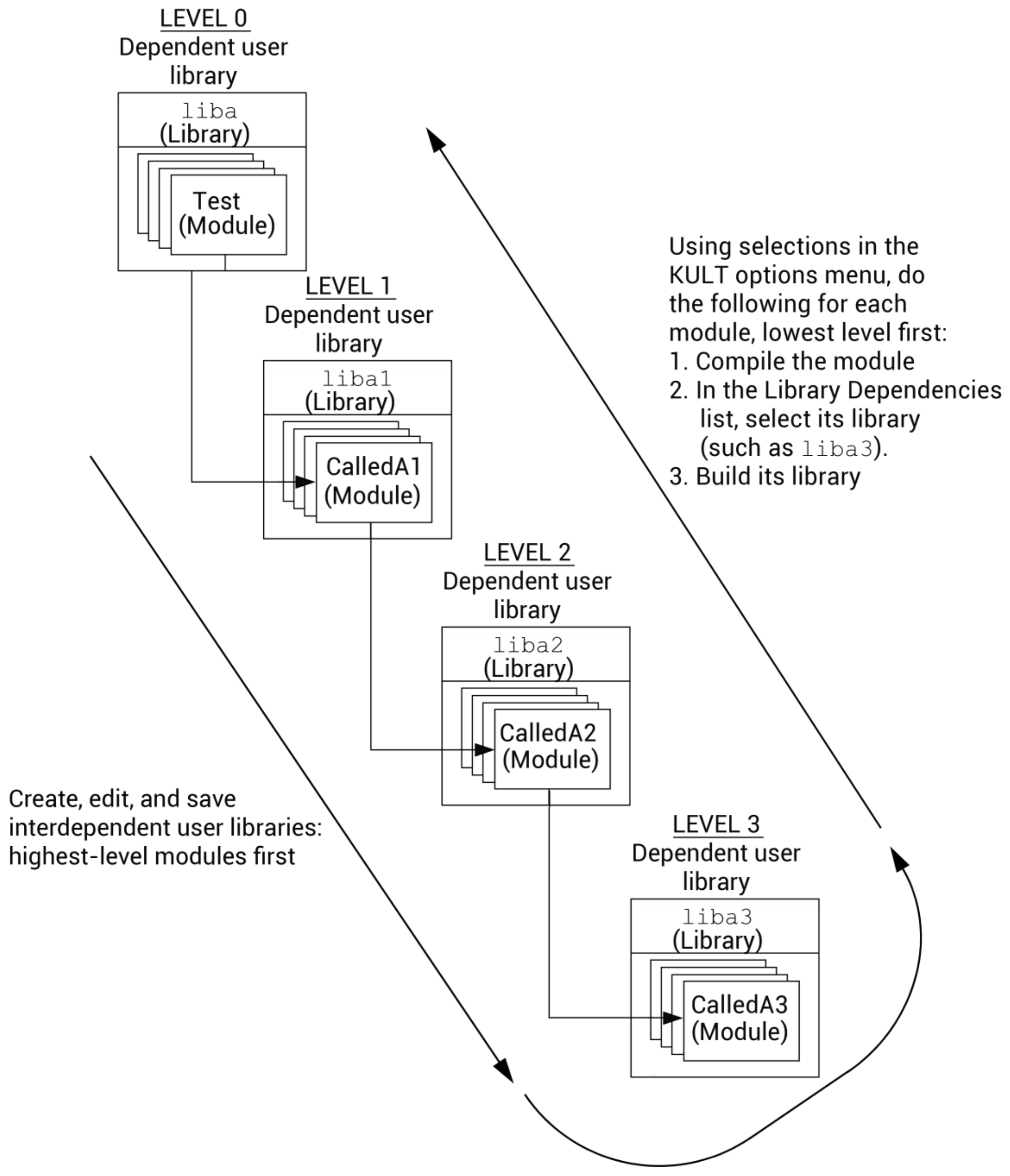
In the following table, the series of coded user modules amplifies the hierarchical dependencies shown in the following figure.

Coded user modules illustrating the use of hierarchical user library dependencies

Hierarchy level	User-library name	User-module name	User-module code
0	`liba`	`Test`	`void Test(void) { printf("In liba, calling CalledA1()\n"); CalledA1(); }`
1	`liba1`	`CalledA1`	`void CalledA1(void) { printf("In liba1, calling CalledA2()\n"); CalledA2(); }`
2	`liba2`	`CalledA2`	`void CalledA2(void) { printf("In liba2, calling CalledA3()\n"); CalledA3(); }`
3	`liba3`	`CalledA3`	`void CalledA3(void) { printf("In liba3, making no calls()\n"); }`

A user module in `liba` calls a user module in `liba1`. In turn, a user module in `liba1` calls a user module in `liba2`. Finally, a user module in `liba2` calls a user module in `liba3`.

Figure 41: Hierarchical design for user library dependencies



Building dependent user libraries in the correct order

When KULT builds a user library that depends on other user libraries, it must link to each of these libraries. For example, when KULT builds `liba`, the following linkages occur: `liba` is linked with `liba1`, the `liba/liba1` pair is linked with `liba2`, the `liba/liba1/liba2` trio is linked with `liba3`, and so on. Therefore, a series of hierarchical dependencies requires a reverse hierarchical build order, starting first with the lowest-level user library. Before building any dependent user library, you must first successfully build each library on which it depends, as illustrated below:

- If `liba` depends on `liba1`, `liba` cannot successfully build until `liba1` has been built.
- If, additionally, `liba1` depends on `liba2`, both `liba` and `liba1` cannot successfully build until `liba2` has been built.
- Finally, if `liba2` depends on `liba3`, then the three higher level user libraries (`liba`, `liba1`, and `liba2`) cannot successfully build until `liba3` has been built.

The following procedure illustrates the correct reverse build order for the dependencies shown in the table and figure in Structuring dependencies hierarchically (on page 3-10). This is a general procedure based on the assumption that each of the interdependent user modules are newly created or were edited since the last build. You do not need to repeat builds that are already complete up to a given level of dependency.

Build the Level 3 user module and user library:

1. Build the saved `CalledA3` user module, which is in the `liba3` user library (in the KULT **Options** menu, select **Build**).
2. Build the `liba3` user library (in the KULT **Options** menu, select **Build**).

Build and set dependencies for the Level 2 user module and user library:

1. Build the saved `CalledA2` user module, which is in the `liba2` user library.
2. Select **Options > Library Dependencies**.
3. Select `liba3` from the Library Dependencies list box.
4. Select **Apply**.
5. Build the `liba2` user library.

Build and set dependencies for the Level 1 user module and user library:

1. Build the saved `CalledA1` user module, which is in the `liba1` user library.
2. Select **Options > Library Dependencies**.
3. Select **liba2** from the Library Dependencies list box.
4. Select **Apply**.
5. Build the `liba1` user library.

Build and set dependencies for the Level 0 user module and user library:

1. Build the saved Test user module, which is in the `liba` user library.
2. Select **Options > Library Dependencies**.
3. Select **liba1** from the Library Dependencies list box.
4. Select **Apply**.
5. Build the `liba` user library.

This reverse hierarchical build order results in a linking scheme that satisfies the dynamic linking requirements of Microsoft® Windows®.

Formatting user module help for the Clarius Help pane

If your user module includes a help description, but it is not set up for HTML, when you create a UTM in Clarius, the Help pane displays the Open UTM Comments button. If you select this button, text from the Description tab in KULT is displayed in an ASCII browser dialog.

You can set up this help to display as formatted HTML in the Help pane using PHP Markdown Extra tools. On the first line of the description, add the following stylesheet and MarkdownExtra code:

```
<!--MarkdownExtra-->

   <link rel="stylesheet" type="text/css"
   href="http://clariusweb/HelpPane/stylesheet.css">
```

In order to see the help in Clarius, you must build the UTM and rebuild the library after entering the Markdown code.

To format the text, you can use some of the following options:

- **Create a first level heading:** Place `=====` under a line to center and bold the line. (You can use any number of `=` characters.)
- **Create a second level heading:** Place `--------` under a line to bold the line. (You can use any number of `-` characters.)
- **To create list:** Insert a blank line before the start of the list, then use 1., 2., and so on to number each item in the list.
- **Italicize text:** Place `*` before and after the text to be italicized.
- **Display text in a fixed-width font:** Put six spaces before each line of the text or use four tilde characters (`~~~~`) before and after the lines of text.

You can make changes to the `.c` file of the user module with KULT or a text editor. After saving changes, to view the changes, select another project tree object and then return to the UTM.

An example of the code entered in the Description tab is shown in Documenting the user module (on page 2-9). An example of the result in the Help pane in Clarius is shown in Checking the user module (on page 2-12).

For information on additional formatting options, refer to the PHP Markdown Extra website of Michel Fortin (michelf.ca/projects/php-markdown/extra/).

PHP Markdown Lib Copyright © 2004-2015 Michel Fortin (michelf.ca/). All rights reserved.

Based on Markdown. Copyright © 2003-2005 John Gruber, Daring Fireball (daringfireball.net/). All rights reserved.

Creating project prompts

KULT provides user modules that you can use to create dialogs to pause a test sequence with a prompt. These dialogs are available as user modules, shown in the following table.

You define the text message for the prompt. When one of these user modules is run, the test sequence pauses. The test sequence continues when a button on the dialog is selected.

Winulib user library

User module	Description
`AbortRetryIgnoreDialog`	Pause test sequence with a prompt to Abort, Retry or Ignore
`InputOkCancelDialog`	Pause test sequence for an input prompt; enter input data (OK) or Cancel
`OkCancelDialog`	Pause test sequence with a prompt to continue (OK) or Cancel
`OkDialog`	Pause test sequence with a prompt to continue (OK)
`RetryCancelDialog`	Pause test sequence with a prompt to Retry or Cancel
`YesNoCancelDialog`	Pause test sequence with a Yes, No, or Cancel decision prompt
`YesNoDialog`	Pause test sequence with a Yes or No decision prompt

Using dialog boxes

The `Winulib` user library has user modules for six action or decision dialogs and one input dialog. The dialog, with example prompts, are shown in Dialog formats (on page 3-16).

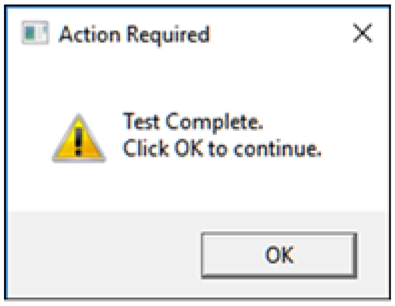
The text message for a prompt is entered by the user into the user module. See “Winulib user-library reference” in the *Model 4200A-SCS Clarius User's Manual* for details on the user modules.

NOTE

An example using the OK dialog is provided in Dialog test examples (on page 3-17).

Dialog formats

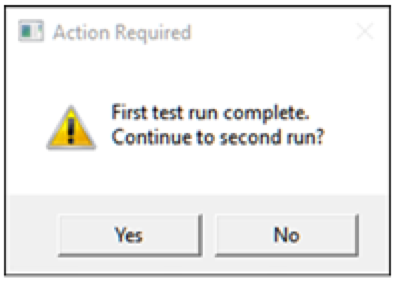
The OK dialog in the following figures has only one button. You can use this dialog to pause a test sequence to make an announcement (for example, "Test Finished"), or prompt for an action (for example, "Connect 590 to DUT"). When OK is selected, the test sequence continues.



Action Required

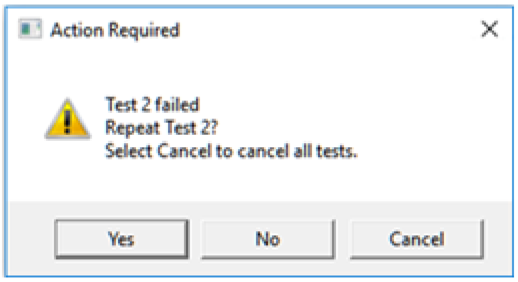
Test Complete.
Click OK to continue.

The other dialogs have two or three buttons, as shown in the following examples. When a button on a dialog is selected, a status value that corresponds to that button is placed in the Analyze sheet for the action. If there are input parameters, the entries for the input parameters are placed in the Analyze sheet. You can pass a parameter value into a user-created routine.



Action Required

First test run complete.
Continue to second run?


Action Required

Test 2 failed
Repeat Test 2?
Select Cancel to cancel all tests.

To pass parameters, the dialog user module must be called from another user-created user module that is designed for parameter passing. A parameter that is in the Analyze sheet is passed to a routine in the user-created user module to perform the appropriate operation or action.

NOTE

An example to demonstrate parameter passing is provided in Dialog test examples (on page 3-17).

Dialog test examples

The following examples demonstrate how you can use dialogs in a test sequence.

Example: Announce end of test

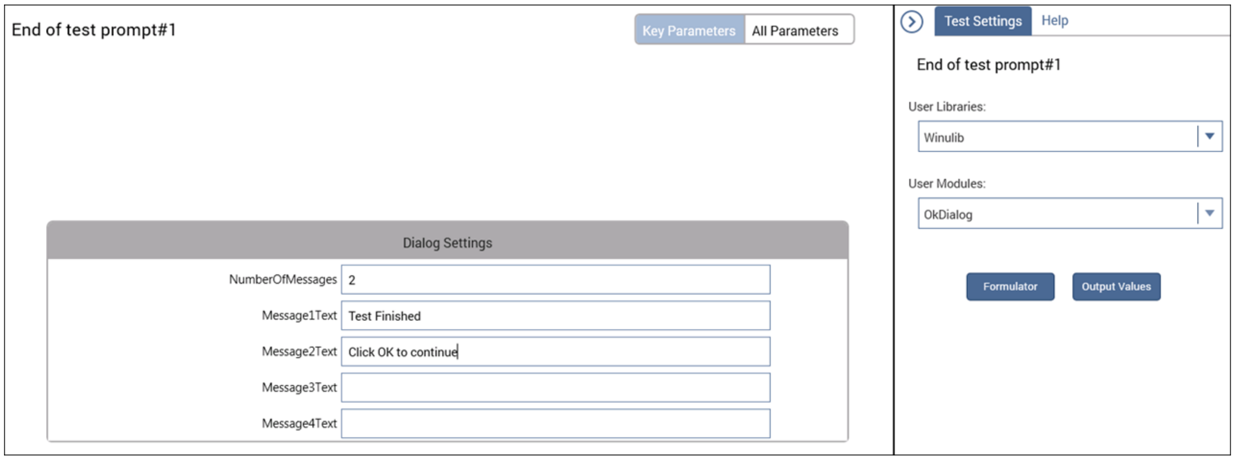
For this example, you will create a user test module (UTM) that uses the OK dialog user module. This dialog announces the end of a test sequence. You can use this UTM in any project at the end of any test sequence.

To create an end-of-test announcement:

1. In the Clarius project tree, select the last test. The announcement occurs after this test.
2. Choose **Select**.
3. Select the **Tests** tab.
4. For the Custom Test, select **Choose a test from the pre-programmed library (UTM)**.
5. Drag **Custom Test** to the project tree. The test has a red triangle next to it to indicate that it is not configured.
6. Select **Rename**.
7. Enter the name **End of test prompt**.
8. Select **Configure**.
9. In the Test Setting pane on the right, set the User Libraries to **Winulib**.

10. Set User Modules to **OkDialog**.
11. For NumberOfMessages, enter **2**.
12. For Message1Text, enter **Test Finished**.
13. For Message2Text, enter **Click OK to continue**. An example is shown in the following figure.

Figure 42: New UTM using OkDialog user module



End of test prompt#1

Dialog Settings

NumberOfMessages	2
Message1Text	Test Finished
Message2Text	Click OK to continue
Message3Text	
Message4Text	

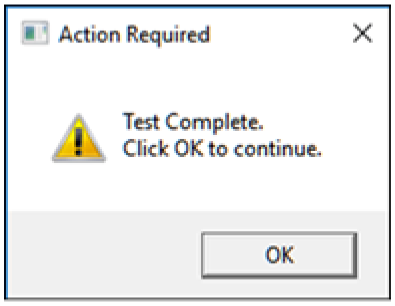
End of test prompt#1

User Libraries: Winulib

User Modules: OkDialog

14. Select **Save**.

When you run the test sequence, the end of test dialog displayed, as shown in the following figure. Select **OK** to continue.



Action Required

Test Complete.
Click OK to continue.

KULT Extension for Visual Studio Code

In this section:

Introduction

The Keithley KULT Extension for Visual Studio Code gives you the ability to write, compile, and debug user libraries outside of KULT. Combining the user-friendly Visual Studio Code editor with KULT creates an integrated development environment (IDE).

This section describes how to download, install, and set up Visual Studio Code and the KULT Extension.

You can use the KULT Extension for Visual Studio Code on a computer with Clarius V1.8 or higher installed. All features for the KULT Extension are available on the computer version of Clarius except the debugging tool. Installation and setup instructions are the same on the 4200A and the computer.

NOTE

The documentation in this section was verified against Visual Studio Code version 1.71.

Installation

You can install Visual Studio Code and the KULT Extension with or without a connection to the internet on the instrument.

These instructions provide information on installing the KULT Extension for the first time and for updating it if Clarius was reinstalled.

Download Visual Studio Code

If you cannot connect to the internet from the instrument, use another computer to download Visual Studio Code.

To download Visual Studio Code:

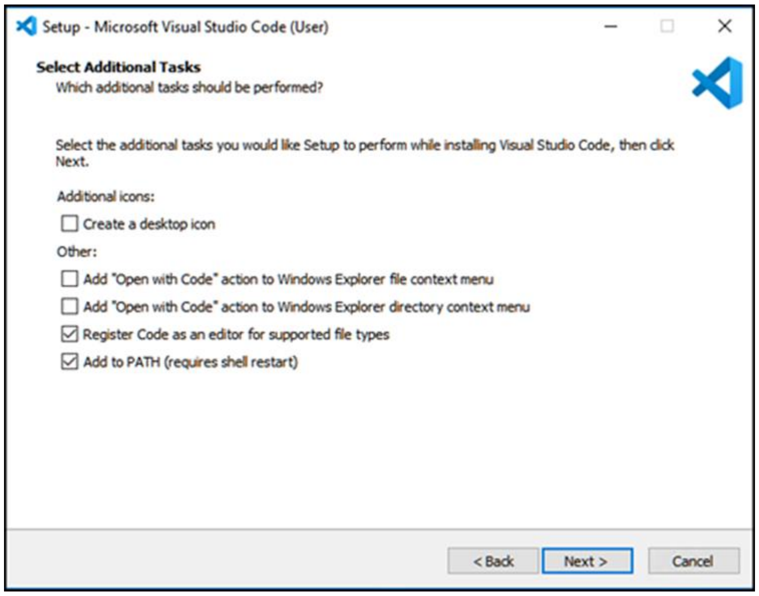
1. Go to the Visual Studio Code download site (code.visualstudio.com/download).
2. Download the Windows User Installer, either 32-bit or 64-bit.
3. If you are downloading from another computer, copy the installation file to a USB flash drive.

Install Visual Studio Code

To install Visual Studio Code:

1. If you downloaded the installation files to a USB flash drive, copy the files to the instrument.
2. Start the installer.
3. Complete the installation wizard.
4. On the Select Additional Tasks dialog, select **Add to PATH (requires shell restart)**. This allows Visual Studio Code to be called from the command line.
5. Make other selections as needed and complete the wizard.

Figure 43: Select additional tasks



Setup - Microsoft Visual Studio Code (User)

Select Additional Tasks

Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Visual Studio Code, then click Next.

Additional icons:

- [ ] Create a desktop icon

Other:

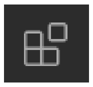
- [ ] Add "Open with Code" action to Windows Explorer file context menu
- [ ] Add "Open with Code" action to Windows Explorer directory context menu
- [x] Register Code as an editor for supported file types
- [x] Add to PATH (requires shell restart)

Install extensions with an internet connection

To install Visual Studio Code extensions:

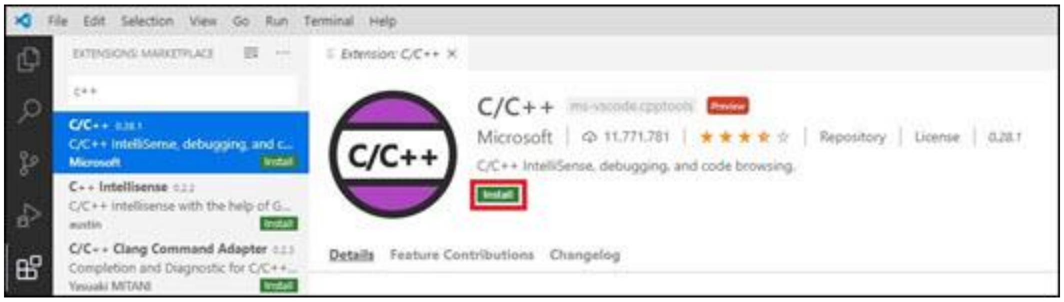
1. From Visual Studio Code, select the Extensions icon in the left navigation. The Extensions pane opens.

Figure 44: Extensions icon



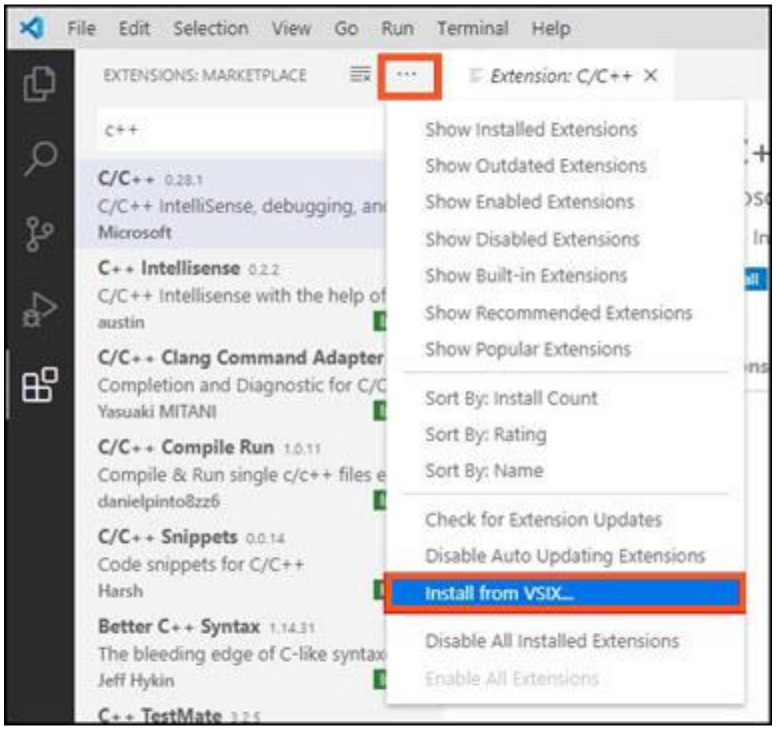
2. Search for `C++` in the Marketplace and select **C/C++**.
3. Select **Install**.

Figure 45: Install the C/C++ Extension


File Edit Selection View Go Run Terminal Help
EXTENSIONS: MARKETPLACE
Extension: C/C++
c++
C/C++ 0.28.1
C/C++ IntelliSense, debugging, and c...
Microsoft
Install
C++ Intellisense 0.2.2
C/C++ Intellisense with the help of G...
austin
Install
C/C++ Clang Command Adapter 0.2.3
Completion and Diagnostic for C/C++...
Yasuaki MITANI
Install
C/C++
C/C++
Microsoft
11,771,781
Repository
License
0.28.1
C/C++ IntelliSense, debugging, and code browsing.
Install
Details
Feature Contributions
Changelog

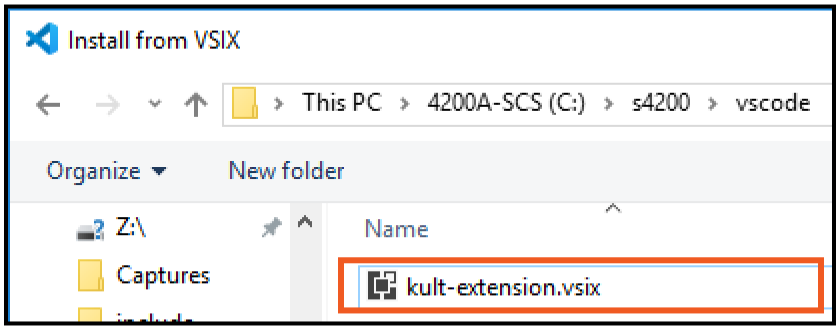
4. At the top of the Extensions: Marketplace pane, select **...** and select **Install from VSIX**.

Figure 46: Install from VSIX



5. Select `C:\s4200\vscode\kult-extension.vsix`.
6. Select **Install**. This installs the KULT Extension to Visual Studio Code.

Figure 47: Install the KULT Extension to Visual Studio Code



7. Close Visual Studio Code and reopen to complete the installation and enable all extensions.
8. Continue to Set up Visual Studio Code for Library Development (on page 4-9).

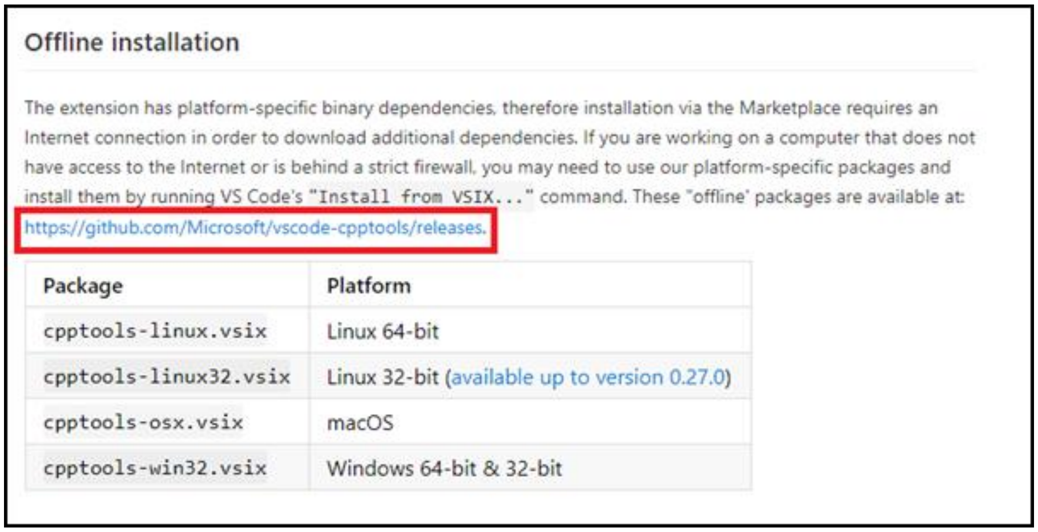
Install extensions without an internet connection

If you do not have an internet connection, you need to use another computer to go the Visual Studio Marketplace to download the Microsoft C/C++ Extension.

To download the Microsoft C/C++ Extension:

1. Go to the C/C++ page of the Visual Studio Marketplace marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools.
2. Scroll down to **Offline Installation** and select the link `https://github.com/Microsoft/vscode-cpptools/releases`.

Figure 48: Offline installation options



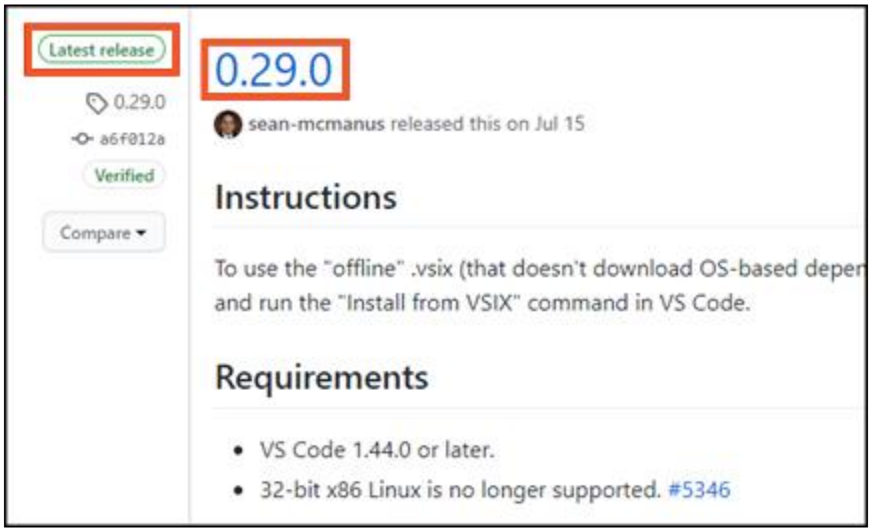
Offline installation

The extension has platform-specific binary dependencies, therefore installation via the Marketplace requires an Internet connection in order to download additional dependencies. If you are working on a computer that does not have access to the Internet or is behind a strict firewall, you may need to use our platform-specific packages and install them by running VS Code's "Install from VSIX..." command. These "offline" packages are available at: https://github.com/Microsoft/vscode-cpptools/releases.

Package	Platform
cpptools-linux.vsix	Linux 64-bit
cpptools-linux32.vsix	Linux 32-bit (available up to version 0.27.0)
cpptools-osx.vsix	macOS
cpptools-win32.vsix	Windows 64-bit & 32-bit

3. Find the version with the Latest Release tag that is verified. Select the version number to view the files. An example is shown in the following graphic.

Figure 49: Example of a version tagged with Latest Release and Verified



0.29.0

sean-mcmanus released this on Jul 15

Instructions

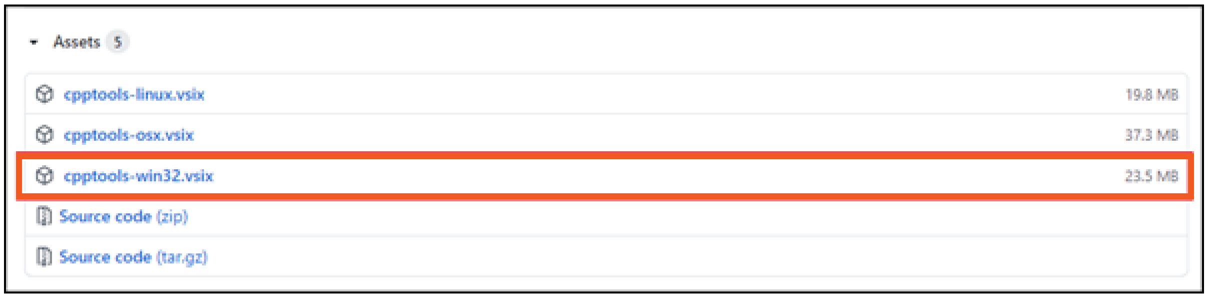
To use the "offline" .vsix (that doesn't download OS-based deper
and run the "Install from VSIX" command in VS Code.

Requirements

- VS Code 1.44.0 or later.
- 32-bit x86 Linux is no longer supported. #5346

4. Download `cpptools-win32.vsix`.

Figure 50: Microsoft C/C++ extension



Assets 5

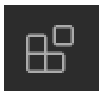
cpptools-linux.vsix	19.8 MB
cpptools-osx.vsix	37.3 MB
cpptools-win32.vsix	23.5 MB
Source code (zip)	
Source code (tar.gz)	

5. Copy the file to a USB drive.

Install the Microsoft C/C++ Extension on the 4200A-SCS:

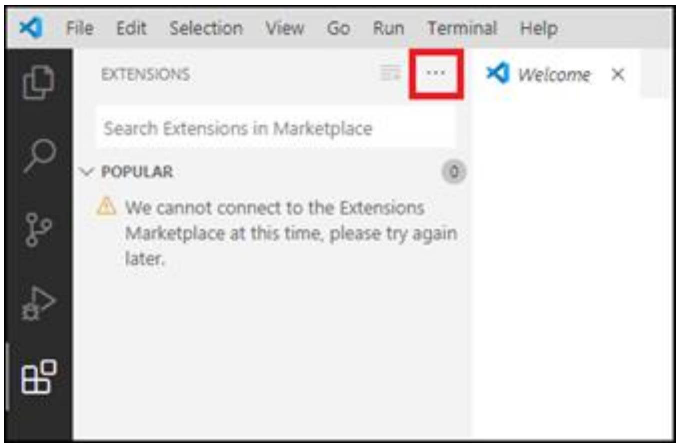
1. Copy `cpptools-win32.vsix` from the USB drive to the `C:\s4200\vscode` folder on the 4200A-SCS.
2. Open Visual Studio Code.
3. Select the Extensions icon in the left navigation. The Extensions pane opens.

Figure 51: Extensions icon



4. Select **…** at the top of the Extensions pane.
5. Select **Install from VSIX**.
6. Select the `cpptools-win32.vsix` file and select **Install**.

Figure 52: Install from VSIX with no internet connection



We cannot connect to the Extensions Marketplace at this time, please try again later.

7. Select **…** at the top of the Extensions pane again.
8. Select **Install from VSIX**.
9. Select the `kult-extensions.vsix` file and select **Install**.
10. Close Visual Studio Code and reopen to complete the installation and enable all extensions.
11. Continue to Set up Visual Studio Code for Library Development (on page 4-9).

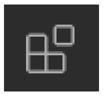
Updating the KULT Extension after installing Clarius

If you installed a new version of Clarius, you must uninstall and reinstall the KULT Extension.

To uninstall the KULT Extension:

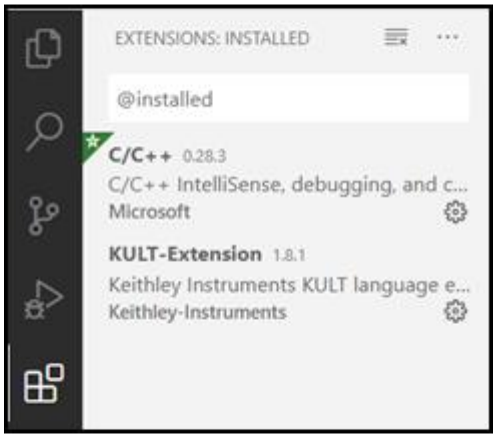
1. Open Visual Studio Code.
2. From Visual Studio Code, select the Extensions icon in the left navigation. The Extensions pane opens.

Figure 53: Extensions icon



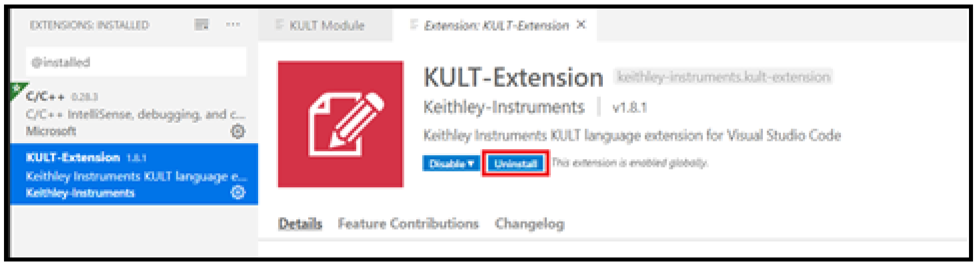
3. Select the **KULT Extension**.

Figure 54: Visual Studio Code Extension Marketplace



4. Select **Uninstall**.

Figure 55: Uninstall the KULT Extension

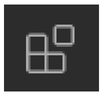

EXTENSIONS: INSTALLED
@installed
C/C++ 0.28.3
C/C++ IntelliSense, debugging, and c...
Microsoft
KULT-Extension 1.8.1
Keithley Instruments KULT language e...
Keithley-Instruments
KULT Module
Extension: KULT-Extension
KULT-Extension
keithley-instruments.kult-extension
Keithley-Instruments
v1.8.1
Keithley Instruments KULT language extension for Visual Studio Code
Disable
Uninstall
This extension is enabled globally.
Details
Feature Contributions
Changelog

5. Close Visual Studio Code.

To reinstall the KULT Extension:

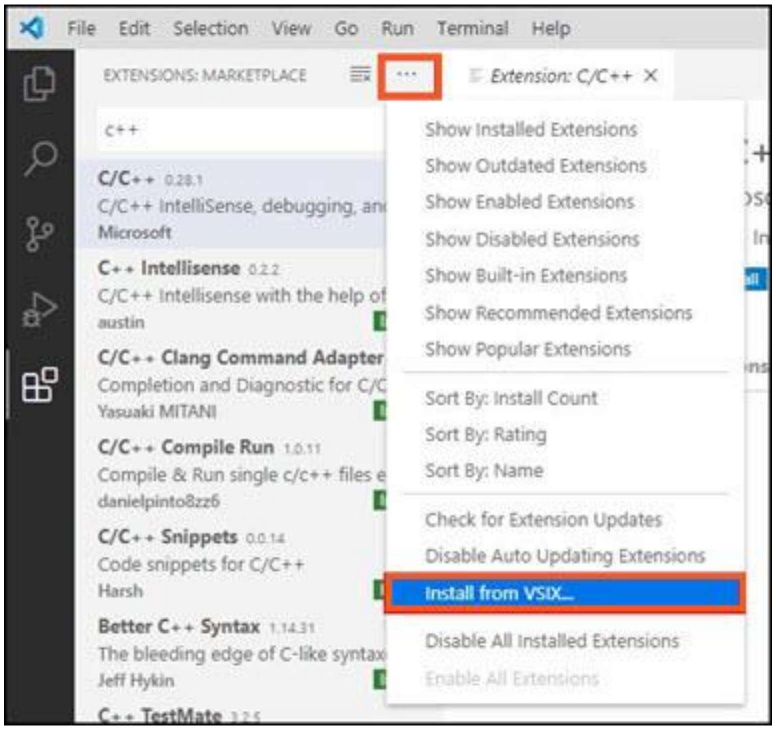
1. From Visual Studio Code, select the Extensions icon in the left navigation. The Extensions pane opens.

Figure 56: Extensions icon



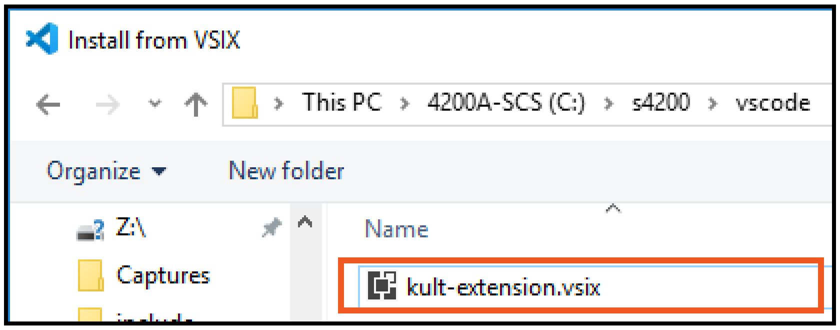
2. At the top of the Extensions: Marketplace pane, select **...** and select **Install from VSIX**.

Figure 57: Install from VSIX



3. Select `C:\s4200\vscode\kult-extension.vsix`.
4. Select **Install**.

Figure 58: Install the KULT Extension to Visual Studio Code



5. Close Visual Studio Code and reopen to complete the installation and enable all extensions.

Setting up Visual Studio Code for library development

Before using Visual Studio Code for developing KULT libraries, you need to open the user library so that you can access all the libraries without switching folders. You can open single user libraries by opening the folder of the user library. On startup, Visual Studio Code reopens the last folder opened.

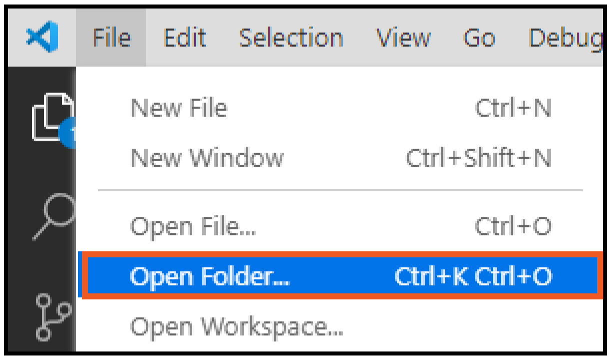
You also need to create the Visual Studio Configuration files before you open a single library. Visual Studio Code configuration files adopt the features of Visual Studio Code to be used with Keithley User Libraries.

Opening the user library in Visual Studio Code

To open the user library folder:

1. Go to **File > Open Folder** to select a folder to open in Visual Studio Code. You must select a valid user library folder to use the KULT Extension.

Figure 59: Open folder dialog



2. Open the usrlib folder `C:\s4200\kiuser\usrlib`.

Creating the Visual Studio Code configuration files

Visual Studio Code configuration files adopt the features of Visual Studio Code to be used with Keithley User Libraries.

The `c_cpp_properties.json` configuration file controls the Intellisense features of the C/C++ Extension from Microsoft, such as compiler-specific syntax checking and header file paths. Intellisense errors may occur if these features are not configured for Keithley user libraries. The errors do not affect compilation or code execution in Clarius, but may make code difficult to troubleshoot.

The `launch.json` file has configuration settings for GNU Debugger (GDB), which is used when debugging. Debugging attaches GDB to the Clarius running process `UTMServer.exe`.

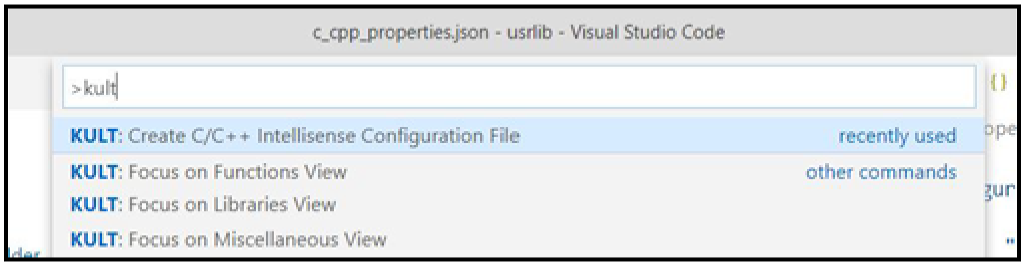
A `.vscode` folder is created with the configuration files in the folder that is open (workspace path). All files in the workspace path reference these configuration files. If the `usrlib` folder is open, the configuration files can be created once, and all the libraries will use them. If you are opening individual libraries, these files need to be created for each library the first time it is opened. Created configuration files can be edited later. The file `settings.json` contains Visual Studio Code workspace level configuration settings.

Create the C/C++ Intellisense configuration file

To create the C/C++ Intellisense configuration file:

1. Open the Command Palette by selecting **View > Command Palette**.
2. Search **KULT** to filter for KULT Extension commands.
3. Select the command **KULT: Create C/C++ Intellisense Configuration File**. This generates the `c_cpp_properties.json` configuration file and places it in the `.vscode` folder in the working directory. The command does not overwrite an existing configuration file.

Figure 60: Generate the c_cpp_properties configuration file



c_cpp_properties.json - usrlib - Visual Studio Code

>kult

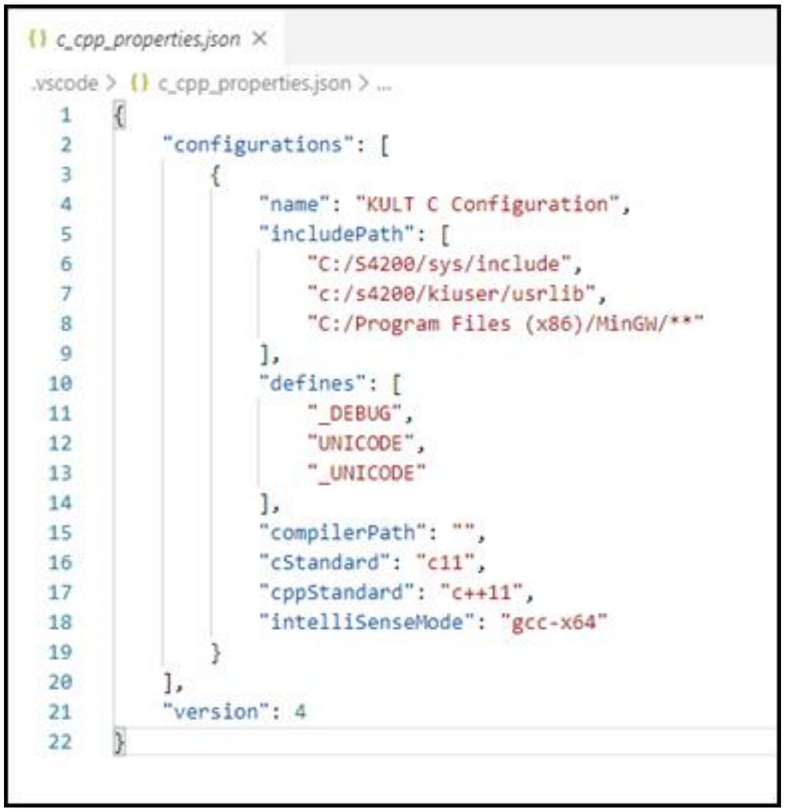
KULT: Create C/C++ Intellisense Configuration File recently used
KULT: Focus on Functions View other commands
KULT: Focus on Libraries View
KULT: Focus on Miscellaneous View

4. Open the KULT side pane by selecting KULT on the left side of the screen.
5. Select a library in the Libraries pane.
6. In the Miscellaneous pane at the bottom of the KULT side pane, select the `c_cpp_properties.json` file. Add paths to header files in the includePath settings. Paths are entered in quotes and separated by commas. This file can be updated at any time.

NOTE

The paths to the include folder, usrlib folder, and compiler are necessary for most user libraries and are automatically entered. Deleting these paths causes Intellisense errors in factory-user libraries.

Figure 61: c_cpp_properties


{} c_cpp_properties.json ×
.vscode > {} c_cpp_properties.json > ...

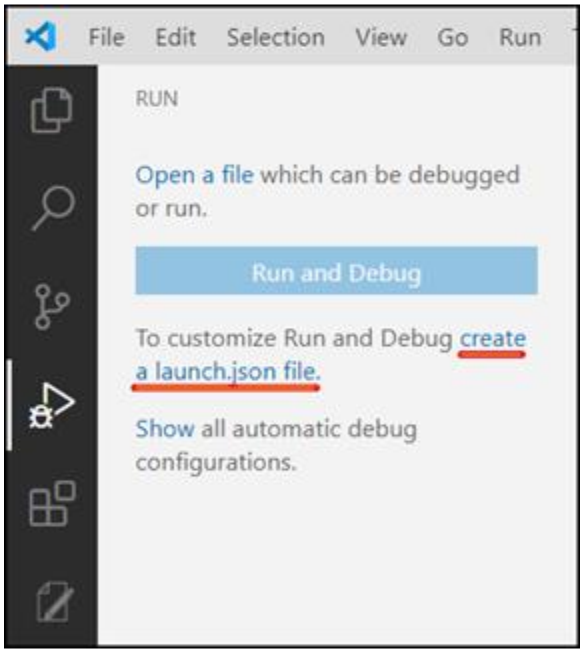
```
{
    "configurations": [
        {
            "name": "KULT C Configuration",
            "includePath": [
                "C:/S4200/sys/include",
                "c:/s4200/kiuser/usrlib",
                "C:/Program Files (x86)/MinGW/**"
            ],
            "defines": [
                "_DEBUG",
                "UNICODE",
                "_UNICODE"
            ],
            "compilerPath": "",
            "cStandard": "c11",
            "cppStandard": "c++11",
            "intelliSenseMode": "gcc-x64"
        }
    ],
    "version": 4
}
```

Create the launch configuration file

To create the launch configuration file:

1. Open the Debug side bar by selecting the debug icon on the left side.
2. Select **Create a launch.json** file.

Figure 62: Create a launch.json file



Open a file which can be debugged or run.

To customize Run and Debug create a launch.json file.

Show all automatic debug configurations.

3. Select **KULT Attach Process**. The `launch.json` file is added with the settings to attach to the UTM Server. The file can be accessed later in the Miscellaneous tab of the KULT side bar.

Figure 63: KULT Attach Process

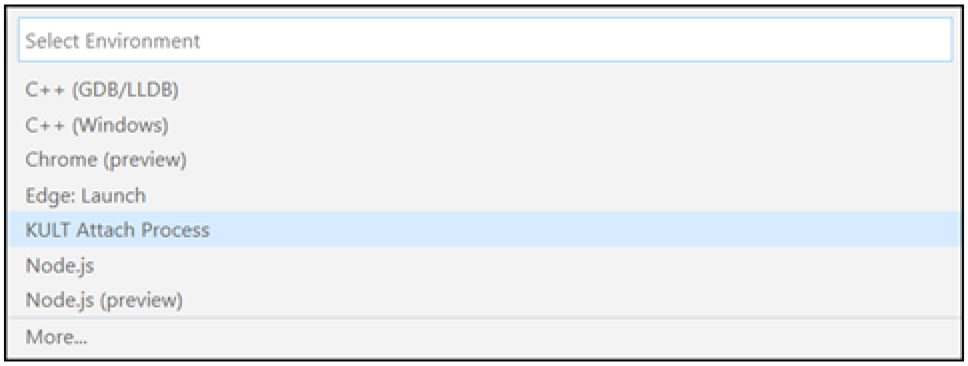
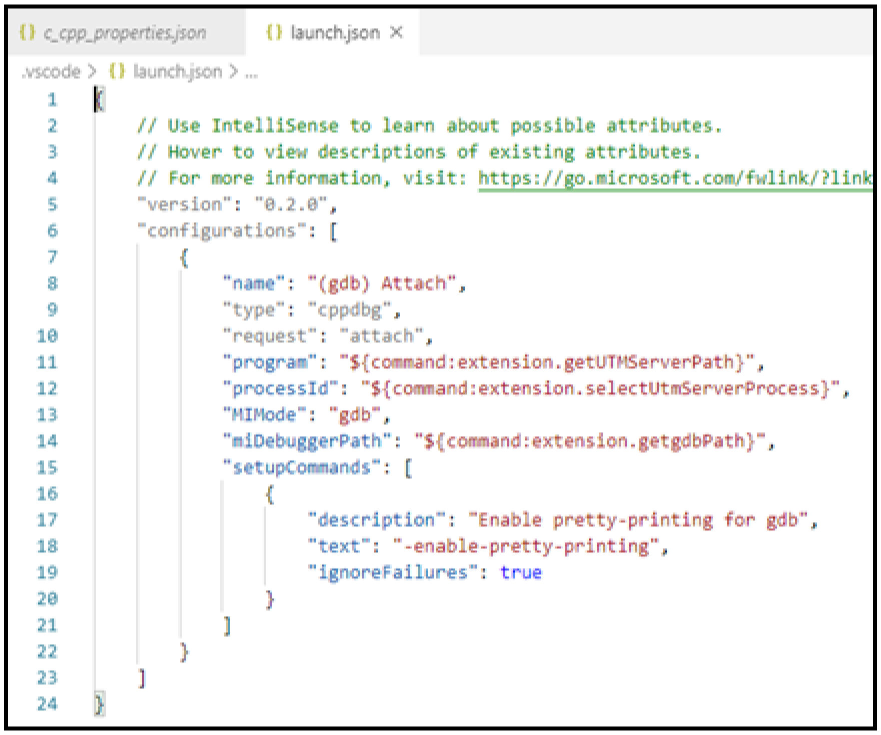


Figure 64: Tab with launch.json



```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?link
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) Attach",
            "type": "cppdbg",
            "request": "attach",
            "program": "${command:extension.getUTMServerPath}",
            "processId": "${command:extension.selectUtmServerProcess}",
            "MIMode": "gdb",
            "miDebuggerPath": "${command:extension.getgdbPath}",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ]
        }
    ]
}
```

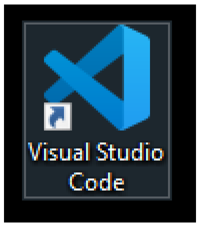
Visual Studio code overview

The following topics describe the features of Visual Studio Code. To learn more about Visual Studio Code as an editor, visit Visual Studio Code (code.visualstudio.com/).

Opening Visual Studio Code

To open Visual Studio Code, select the desktop icon or select Visual Studio Code in the Windows Start Menu.

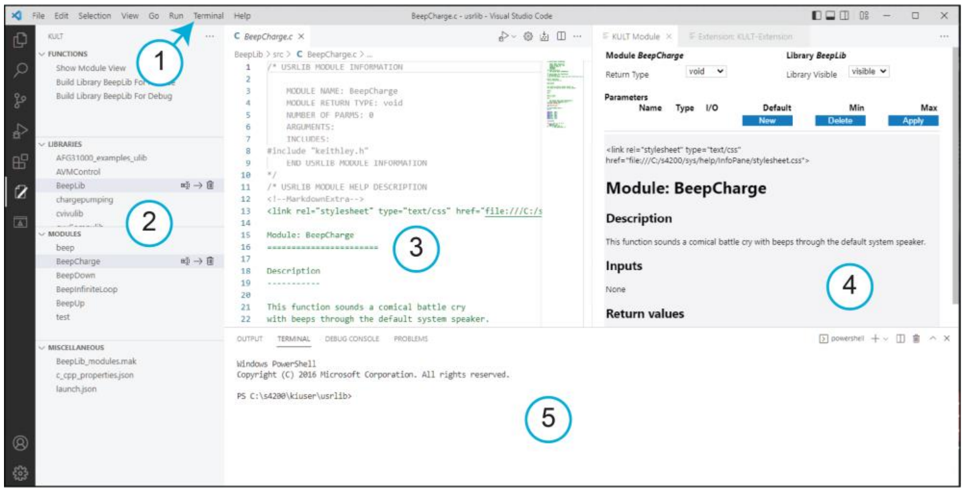
Figure 65: Visual Studio Code



Visual Studio Code user interface

Important parts of the main user interface are labeled in the following figure.

Figure 66: Visual Studio Code user interface components



```
/* USRLIB MODULE INFORMATION

    MODULE NAME: BeepCharge
    MODULE RETURN TYPE: void
    NUMBER OF PARMS: 0
    ARGUMENTS:
    INCLUDES:
#include "keithley.h"
    END USRLIB MODULE INFORMATION
*/
/* USRLIB MODULE HELP DESCRIPTION
<!--MarkdownExtra-->
<link rel="stylesheet" type="text/css" href="file:///C:/s

Module: BeepCharge
=======================

Description
-----------

This function sounds a comical battle cry
with beeps through the default system speaker.
```

Module *BeepCharge* Library *BeepLib*

Return Type: void Library Visible: visible

Parameters

Name	Type	I/O	Default	Min	Max
			New	Delete	Apply

<link rel="stylesheet" type="text/css" href="file:///C:/s4200/sys/help/InfoPane/stylesheet.css">

Module: BeepCharge

Description

This function sounds a comical battle cry with beeps through the default system speaker.

Inputs

None

Return values

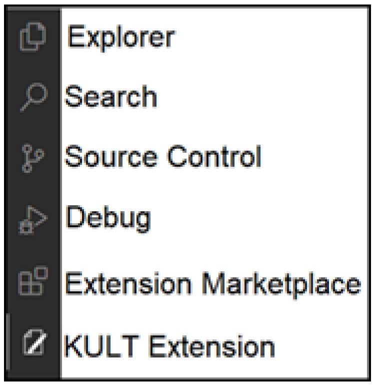
```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\s4200\kiuser\usrlib>
```

1	Terminal menu	Run Task shows all tasks available in Visual Studio Code. Run Build Task displays a subset of the tasks specific to building. KULT build tasks are specific to a library that can be selected by opening a library module.
2	Side bar	Displays views that assist you when editing. You can switch views using the icons in the activity bar next to the side bar.
3	Editor	Displays open files. You can right-click the tabs to change the view and display multiple files.
4	KULT Module	Make changes to the module parameters and code.
5	Panels	Manage user output. See Panels (on page 4-15) for more information.

Activity bar

The Visual Studio Code side bar includes an activity bar that allows you to switch between views and additional context-specific indicators. An example of the activity bar is shown in the following figure.

Figure 67: Visual Studio Code activity bar with KULT Extension



The activity bar includes:

- **Explorer**: Displays all files in the Visual Studio Code working directory.
- **Search**: Search and replace options for open files.
- **Source control**: Not used by the KULT Extension.
- **Debug**: Allows you to monitor variables, threads, and breakpoints during debug mode.
- **Extension marketplace**: Install and uninstall extensions to Visual Studio Code.
- **KULT**: Displays libraries and modules, build functions, and other useful tools for developing libraries. See KULT side bar (on page 4-17) for additional information.

Panels

You can display panels below the editor region. Panels display information to the user, such as output and debug information and errors.

To display panels, select **View > Open View > Panel**.

Panels include:

- **Output**: Certain nonbuild KULT Extension functions, such as Clean Library, provide messages here.
- **Terminal**: Displays output from build tasks in the same format as KULT.
- **Debug Console**: Used for expression evaluation and other tools during debugging.
- **Problems**: Displays various errors found before and during compilation. Select an error message in the Problems panel to display the line of code in the editor.

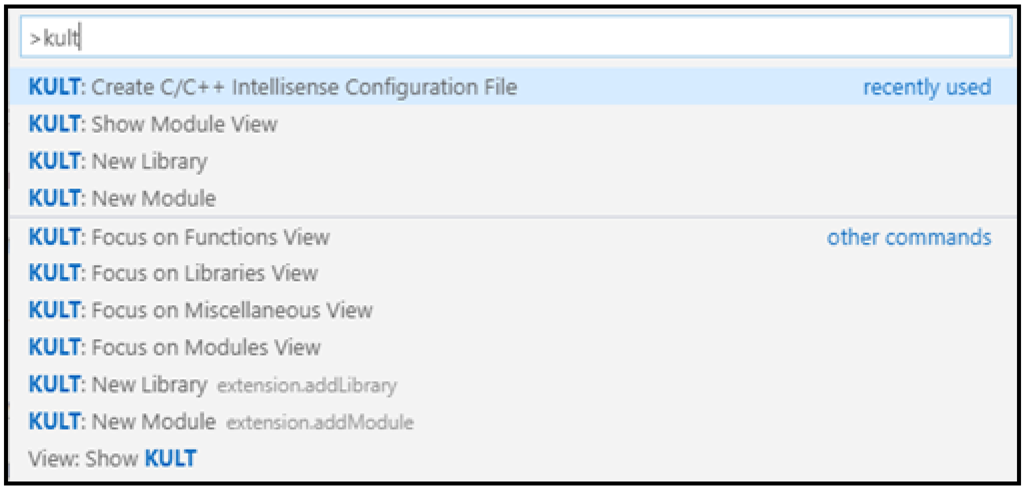
Command Palette

The Command Palette provides access to all Visual Studio Code commands, including the most commonly used commands for the KULT Extension.

To open the Command Palette, select **View > Command Palette** from menu bar.

To display only KULT commands, type **KULT** in the search bar, as shown in the following figure.

Figure 68: Command Palette



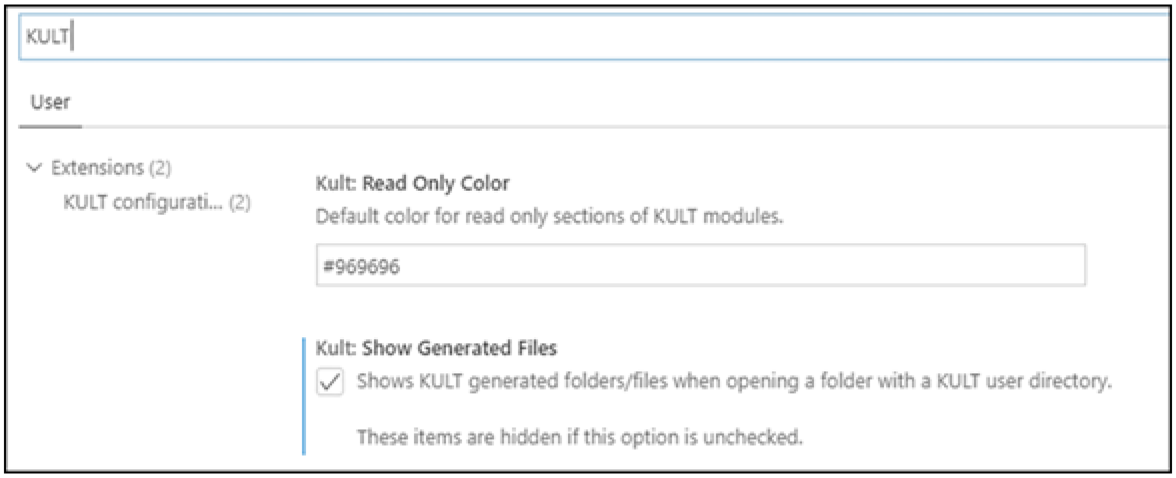
```
>kult
KULT: Create C/C++ Intellisense Configuration File          recently used
KULT: Show Module View
KULT: New Library
KULT: New Module
KULT: Focus on Functions View                              other commands
KULT: Focus on Libraries View
KULT: Focus on Miscellaneous View
KULT: Focus on Modules View
KULT: New Library  extension.addLibrary
KULT: New Module  extension.addModule
View: Show KULT
```

Settings in Visual Studio Code

You can use the settings preferences to personalize Visual Studio Code. To access the settings, select **File > Preferences > Settings**.

To make changes to KULT Extension features, search **KULT** in the settings.

Figure 69: KULT Extension settings



KULT

User

Extensions (2)
KULT configurati... (2)

Kult: Read Only Color
Default color for read only sections of KULT modules.
#969696

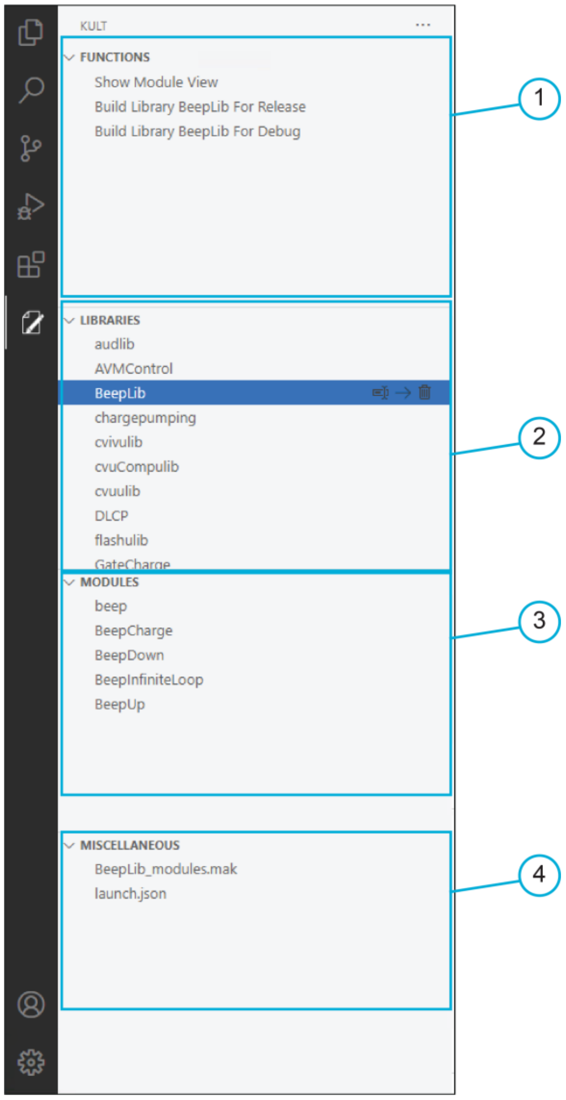
Kult: Show Generated Files
✓ Shows KULT generated folders/files when opening a folder with a KULT user directory.

These items are hidden if this option is unchecked.

KULT side bar

The KULT Extension adds a side bar to Visual Studio Code that enables simplified access to libraries and functions. The following figure shows an example of the KULT side bar. Descriptions of each option are in the table below the figure.

Figure 70: KULT side bar



KULT side bar

1	Functions include options such as opening the KULT module and building a library.
2	Libraries includes all available libraries in alphabetical order.
3	Modules includes all user modules for the select library. Select a module to open it in the editor.
4	Miscellaneous includes files that are useful to library development.

Working with user libraries in Visual Studio Code

This section covers the basics of working with user libraries in the Visual Studio Code KULT Extension.

NOTE

The KULT extension supports restricted mode in Visual Studio Code. If a workspace is untrusted, you cannot compile UTMs. For more information on workspace trust in Visual Studio Code, refer to https://code.visualstudio.com/docs/editor/workspace-trust.

CAUTION

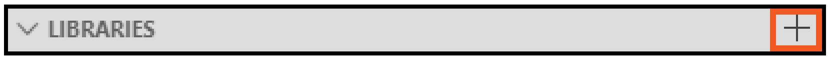
To prevent malicious code execution, do not run a UTM if you have not verified the source.

Creating a new library

To create a new library:

1. In the KULT Extension Libraries side bar, select **+**.

Figure 71: Create a new library



2. Type a name for the library.
3. Press **Enter**.

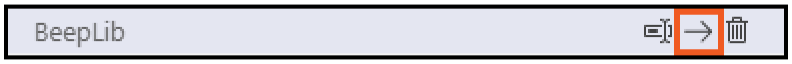
Copying a library

When you copy a library, the user modules for the library are also copied. "Copy" is added to the names of the copied library and user modules.

To copy a library:

1. Select the library.
2. Select the copy icon.

Figure 72: Copy a library or module



3. The copied library must be built before you can use it in Clarius. See Building a library (on page 4-22).

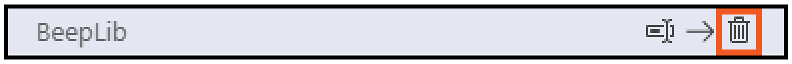
Deleting a library

When you delete a library, all the modules and associated build files in the library are also deleted.

To delete a library:

1. In the Libraries side bar, select the library.
2. Select the delete icon.

Figure 73: Delete a library or module



3. Select **Yes**.

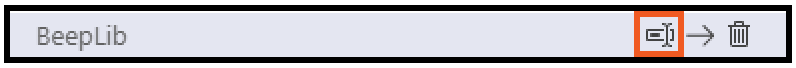
Renaming a library

You can change the name of a library. However, you cannot change the case of the letters in a library name.

To rename a library:

1. Select the library.
2. Select the change name icon.

Figure 74: Rename a library or module



3. Type the new name.
4. Select **Enter**.
5. The renamed library must be built before you can use it in Clarius. See Building a library (on page 4-22).

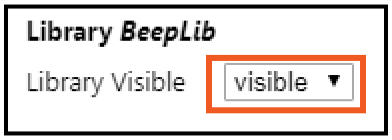
Setting library visibility

You can set a library to be available or unavailable to Clarius. For example, you can hide a library if you want to designate that a user library is only to be called by another user library and is not to be connected to a UTM.

To set the visibility of a library:

1. In the KULT side bar, select a library.
2. Select a module in the library.
3. In the KULT Module tab, change the setting of **Library Visible** as needed.

Figure 75: Select library visibility



Library *BeepLib*

Library Visible: visible

4. Select **Apply**.

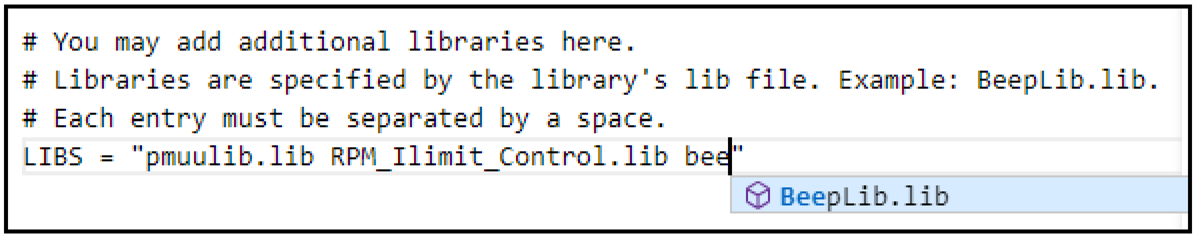
Entering library dependencies and environment variables

Library dependencies allow a user library to call other libraries. You can edit library dependencies directly in the `.mak` file of the library. Library files that are not in the workspace directory, such as third-party libraries, can be added in the `.mak` file. The LIB environment variable of the system must be updated with the path to this library. Users can also update the INCLUDE path environment variable for header files located outside of the workspace directory.

To add a library dependency:

1. Select the library to edit in the KULT side bar.
2. Under Miscellaneous, select *`libName`*`_modules.mak`.
3. For the variable LIBS, type the name of the library between the quotes. To enter multiple libraries, separate the library names with spaces. You can press **Ctrl+Space** to choose from a list of all available libraries. Type the library name to filter the results. Press **Enter** to select a library.

Figure 76: Add a library dependency

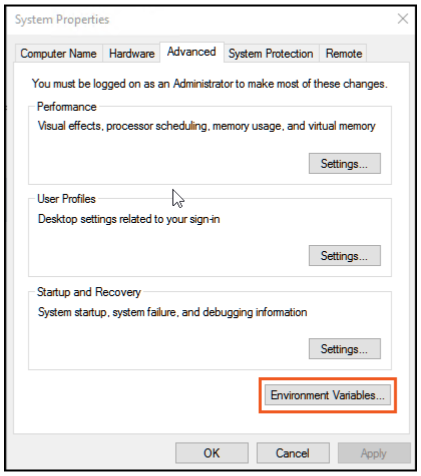


```
# You may add additional libraries here.
# Libraries are specified by the library's lib file. Example: BeepLib.lib.
# Each entry must be separated by a space.
LIBS = "pmuulib.lib RPM_Ilimit_Control.lib bee"
```

To update the system environment variables for external libraries and headers:

1. In the Windows search bar, type **Environment Variables**.
2. Select **Edit the system environment variables**.
3. Select the Advanced tab.
4. Select **Environment Variables.**

Figure 77: Set environmental variables



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

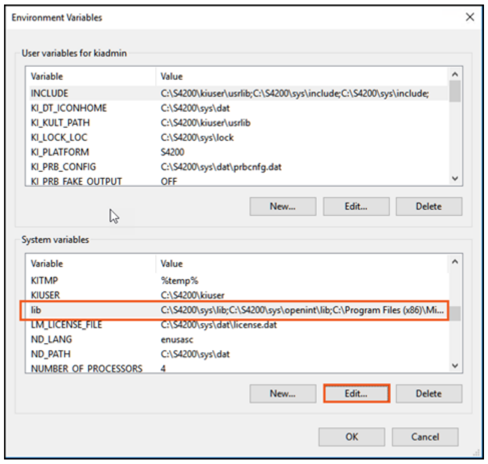
Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your sign-in

Startup and Recovery
System startup, system failure, and debugging information

5. In the System Variables box, select **lib**.
6. Select **Edit**.

Figure 78: Enter environmental variables



Environment Variables

User variables for kiadmin

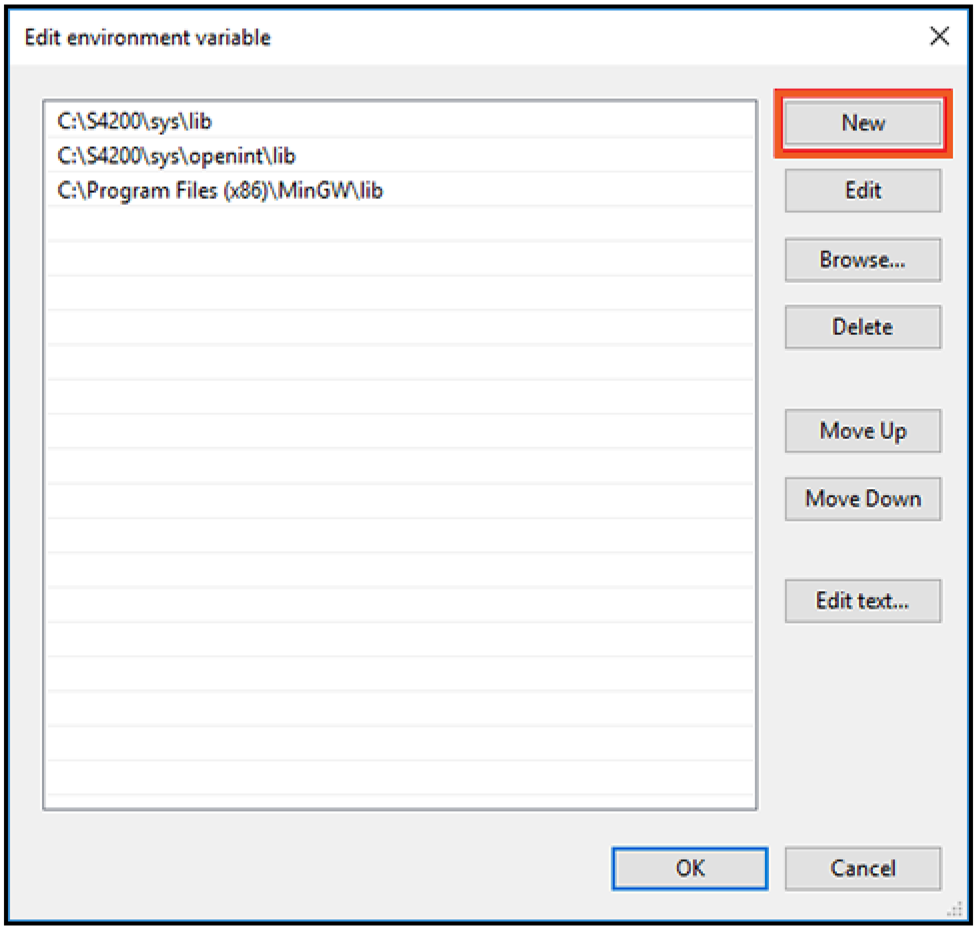
Variable	Value
INCLUDE	C:\S4200\kiuser\usrlib;C:\S4200\sys\include;C:\S4200\sys\include;
KI_DT_ICONHOME	C:\S4200\sys\dat
KI_KULT_PATH	C:\S4200\kiuser\usrlib
KI_LOCK_LOC	C:\S4200\sys\lock
KI_PLATFORM	S4200
KI_PRB_CONFIG	C:\S4200\sys\dat\prbcnfg.dat
KI_PRB_FAKE_OUTPUT	OFF

System variables

Variable	Value
KITMP	%temp%
KIUSER	C:\S4200\kiuser
lib	C:\S4200\sys\lib;C:\S4200\sys\openint\lib;C:\Program Files (x86)\Mi...
LM_LICENSE_FILE	C:\S4200\sys\dat\license.dat
ND_LANG	enusasc
ND_PATH	C:\S4200\sys\dat
NUMBER_OF_PROCESSORS	4

7. Select **New** and enter the path to the external library. You can repeat this process for any header files by selecting the **INCLUDES** variable and entering in the path to the header file.

Figure 79: Edit environment variable



C:\S4200\sys\lib
C:\S4200\sys\openint\lib
C:\Program Files (x86)\MinGW\lib

Building a library

When building a library, you can build for debug or build for release. Building a library for debug creates symbols that the debugger requires to watch variables. If you are not using the debugger, you can build for release, which does not create these symbols.

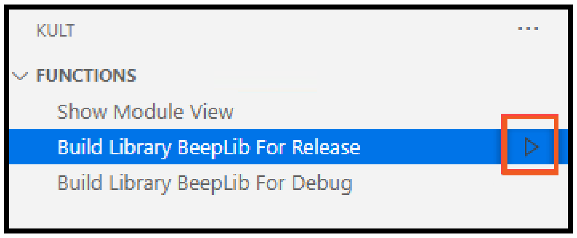
You can build a library from the KULT Extension side bar or from the Terminal menu.

Build a library from the KULT Extension side bar

To build a library from the KULT Extension side bar:

1. Select the library.
2. Under Functions, select the run icon next to `Build Library LibName for Release` or `Build Library LibName for Debug` in the Functions tab of the KULT side bar.

Figure 80: Run icon for Build Library


KULT
...
FUNCTIONS
Show Module View
Build Library BeepLib For Release
Build Library BeepLib For Debug

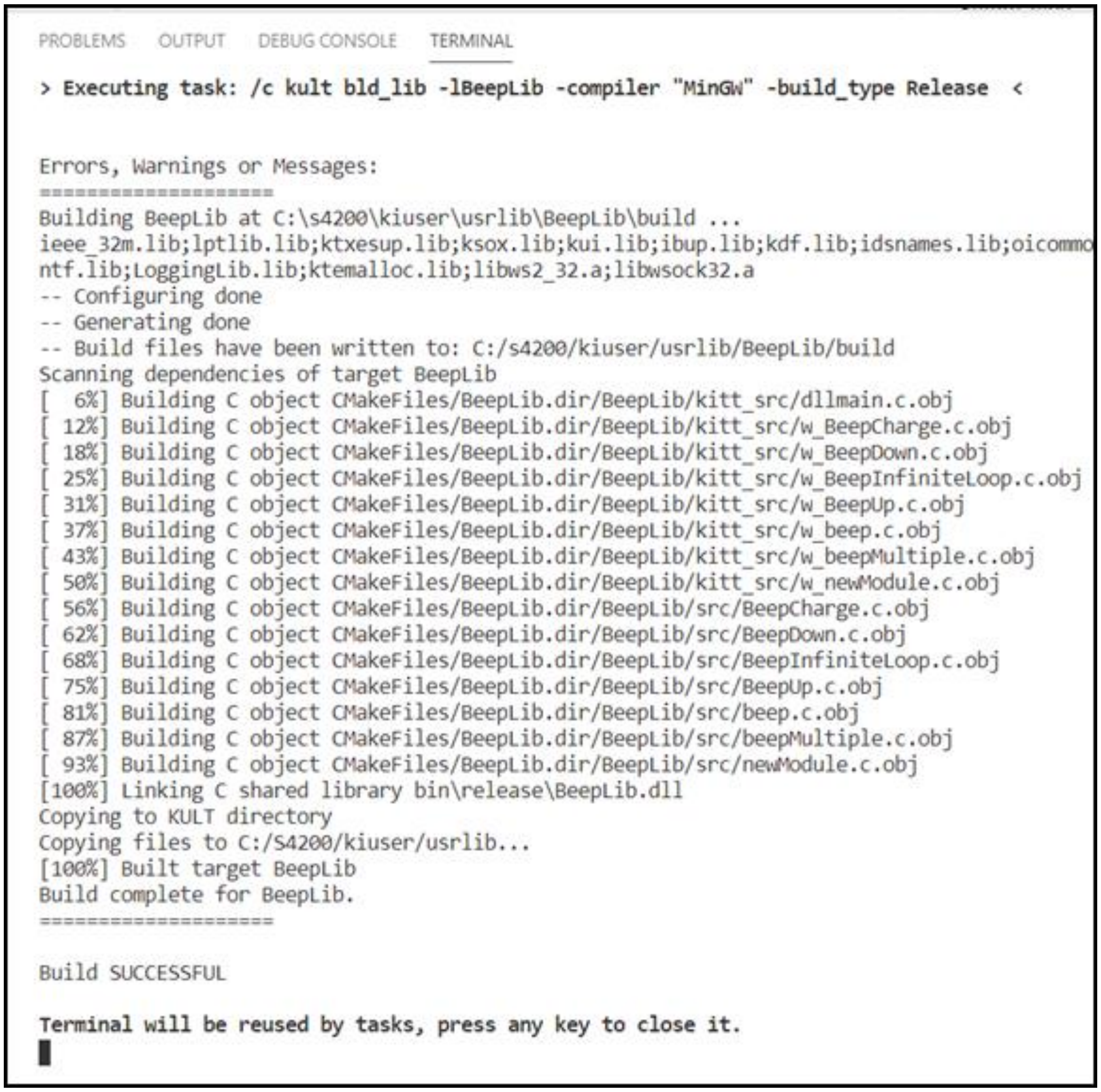
3. Select **Terminal** at the bottom of the screen to view the build status.
4. To view problems with the build, select **Problems**.

Build a library from the Terminal menu

To build a library from the Terminal menu:

1. Select the library.
2. From the **Terminal** menu, select **Run Build Task**.
3. Select the debug or release build option.
4. Select **Terminal** at the bottom of the screen to view the build status.

Figure 81: Build Terminal status



```
> Executing task: /c kult bld_lib -lBeepLib -compiler "MinGW" -build_type Release  <


Errors, Warnings or Messages:
====================
Building BeepLib at C:\s4200\kiuser\usrlib\BeepLib\build ...
ieee_32m.lib;lptlib.lib;ktxesup.lib;ksox.lib;kui.lib;ibup.lib;kdf.lib;idsnames.lib;oicommo
ntf.lib;LoggingLib.lib;ktemalloc.lib;libws2_32.a;libwsock32.a
-- Configuring done
-- Generating done
-- Build files have been written to: C:/s4200/kiuser/usrlib/BeepLib/build
Scanning dependencies of target BeepLib
[  6%] Building C object CMakeFiles/BeepLib.dir/BeepLib/kitt_src/dllmain.c.obj
[ 12%] Building C object CMakeFiles/BeepLib.dir/BeepLib/kitt_src/w_BeepCharge.c.obj
[ 18%] Building C object CMakeFiles/BeepLib.dir/BeepLib/kitt_src/w_BeepDown.c.obj
[ 25%] Building C object CMakeFiles/BeepLib.dir/BeepLib/kitt_src/w_BeepInfiniteLoop.c.obj
[ 31%] Building C object CMakeFiles/BeepLib.dir/BeepLib/kitt_src/w_BeepUp.c.obj
[ 37%] Building C object CMakeFiles/BeepLib.dir/BeepLib/kitt_src/w_beep.c.obj
[ 43%] Building C object CMakeFiles/BeepLib.dir/BeepLib/kitt_src/w_beepMultiple.c.obj
[ 50%] Building C object CMakeFiles/BeepLib.dir/BeepLib/kitt_src/w_newModule.c.obj
[ 56%] Building C object CMakeFiles/BeepLib.dir/BeepLib/src/BeepCharge.c.obj
[ 62%] Building C object CMakeFiles/BeepLib.dir/BeepLib/src/BeepDown.c.obj
[ 68%] Building C object CMakeFiles/BeepLib.dir/BeepLib/src/BeepInfiniteLoop.c.obj
[ 75%] Building C object CMakeFiles/BeepLib.dir/BeepLib/src/BeepUp.c.obj
[ 81%] Building C object CMakeFiles/BeepLib.dir/BeepLib/src/beep.c.obj
[ 87%] Building C object CMakeFiles/BeepLib.dir/BeepLib/src/beepMultiple.c.obj
[ 93%] Building C object CMakeFiles/BeepLib.dir/BeepLib/src/newModule.c.obj
[100%] Linking C shared library bin\release\BeepLib.dll
Copying to KULT directory
Copying files to C:/S4200/kiuser/usrlib...
[100%] Built target BeepLib
Build complete for BeepLib.
====================

Build SUCCESSFUL

Terminal will be reused by tasks, press any key to close it.
█
```

5. To view problems with the build, select **Problems**.

Cleaning a library

Cleaning a library deletes all files that were generated by a build, leaving the source code. This is useful if a build file gets corrupted.

NOTE

When you select **Terminal > Run Task** or **Terminal > Run Build Task**, the build tasks are added to the Run Task history of recently used tasks. This list persists when you close and reopen Visual Studio Code. If the list is too long, you can change the history size using the Manage option on the lower left. Select **Manage > User > Task > Quick Open: History**.

To clean a library:

1. Select the library in the KULT Extension side bar.
2. From the Terminal menu, select **Run Task**.
3. Type **KULT** to limit the list to KULT tasks.
4. Select **KULT: Clean Library “LibName**”.
5. Select **Output** at the bottom of the window to review the actions.

Working with modules in Visual Studio Code

This section covers basics on working with KULT user modules in the Visual Studio Code KULT Extension.

The KULT Extension displays the parameters and description of a module in the editor pane. To display the module in a form, select **Show Module View** from the Functions in the KULT side bar.

NOTE

To view all KULT Extension features, you must open a user library in Visual Studio Code. See Opening the user library directory in Visual Studio Code (on page 4-9) for instructions.

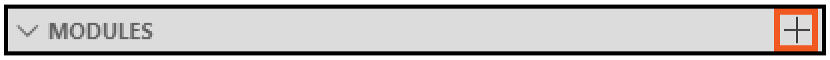
Creating a new user module

All user modules must have unique names to avoid conflicts in library dependencies.

To create a new module:

1. In the KULT Extension side bar, in Libraries, select the library that will contain the module.
2. In Modules, select **+**.

Figure 82: Create a new module



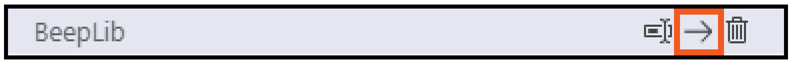
3. Type a name for the new module.
4. Select **Enter**.

Copy a user module

To copy a user module:

1. Select the user module.
2. Select the copy icon.

Figure 83: Copy a library or module



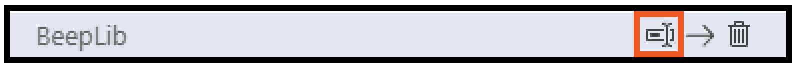
Rename a user module

All user modules must have unique names to avoid conflicts in library dependencies.

To rename a user module:

1. In the KULT Extension side bar, select the module.
2. Select rename icon.

Figure 84: Rename a library or module



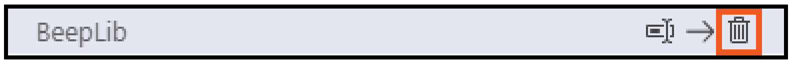
3. Type a name for the new module.
4. Select **Enter**.

Deleting a user module

To delete a module:

1. In the Modules of the sidebar, select the module.
2. Select the delete icon.

Figure 85: Delete a library or module



3. Select **Yes**.

Setting the return type of a user module

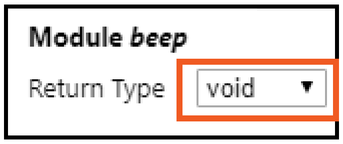
The return type is set for user modules that return a value. The available return types are:

- **char:** Character data
- **float:** Single-precision floating point data
- **double:** Double-precision data
- **int:** Integer data
- **long:** 32-bit integer data
- **void:** No return value

To set the return type of a user module:

1. In the KULT Extension Libraries side bar, select the Library that contains the module.
2. Under Modules, select the user module.
3. In the KULT Module, select the **Return Type**.

Figure 86: Select the return type of a user module



Module *beep*

Return Type: void

4. Select **Apply**.

Including header files

Header files are included in the code before the main module function.

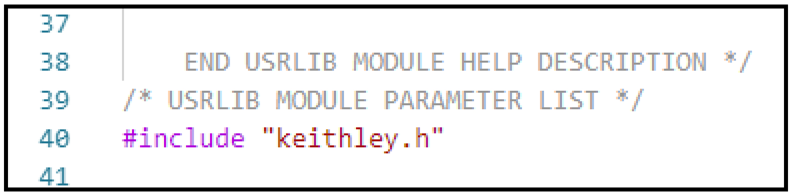
NOTE

Intellisense errors may appear in the Problems tab because paths to header files are not listed in the Intellisense configuration file, `c_cpp_properties.json`. These errors do not affect compilation and can be ignored, or you can follow the instructions below to prevent them.

To add a header file to a module:

1. In the KULT Extensions side bar, select the library that contains the user module.
2. Select the module. The module is displayed in the editor.
3. Add the header file directly below the comment **USRLIB MODULE PARAMETER LIST** using the format `#include "`*`headerName`*`.h"`. An example is shown in the following figure.

Figure 87: Add a header file to a module

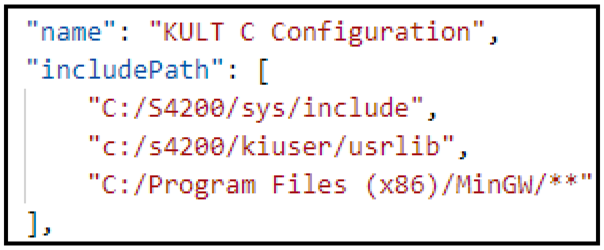


```

    END USRLIB MODULE HELP DESCRIPTION */
/* USRLIB MODULE PARAMETER LIST */
#include "keithley.h"

```

To remove Intellisense header file errors:

1. Create the `c_cpp_properties.json` file if it does not already exist in the `.vscode` folder. See Creating the Visual Studio Code configuration (on page 4-9) files for instructions.
2. If the file already exists, or creating the file did not remove the errors, select the `c_cpp_properties.json` file in the KULT Extension Miscellaneous side bar.
3. Add the header file to the `includePath` setting. File paths must be enclosed in quotes and separated by commas.

Figure 88: Remove Intellisense header file errors



```
"name": "KULT C Configuration",
"includePath": [
    "C:/S4200/sys/include",
    "c:/s4200/kiuser/usrlib",
    "C:/Program Files (x86)/MinGW/**"
],
```

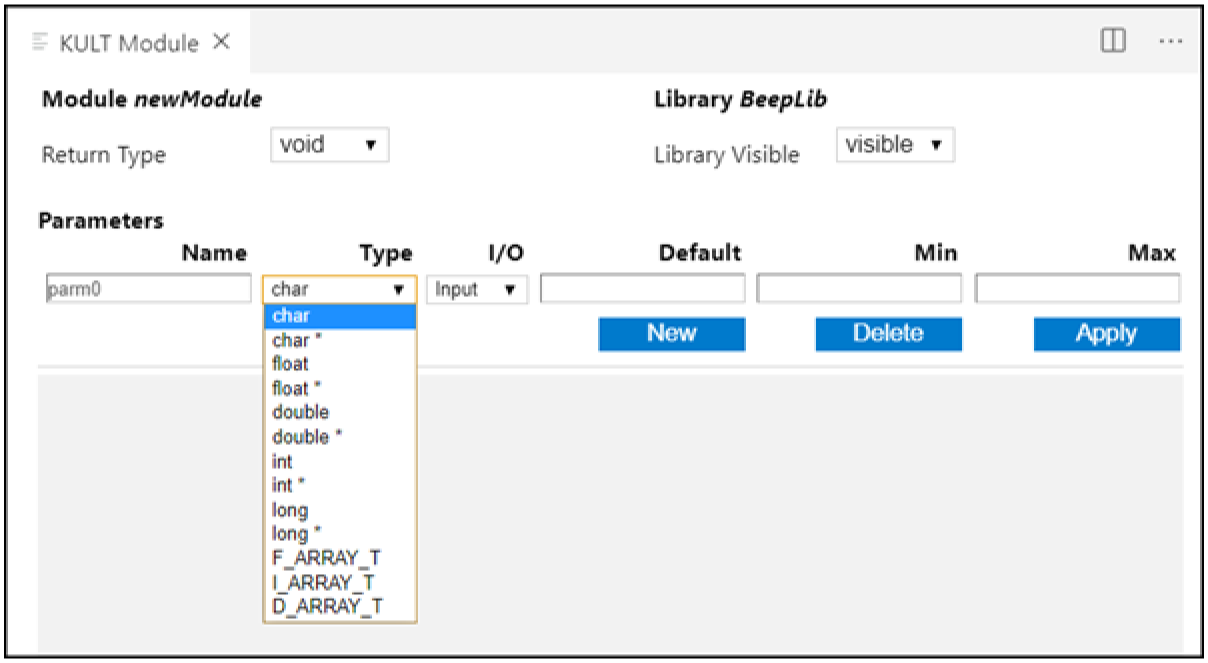
NOTE

You may need to add header files to system environment variables in other places on the system. See Entering Library Dependencies and Environment Variables (on page 4-20) for more information.

Editing module parameters

You can change user module parameters in the KULT Module.

Figure 89: Edit user module parameters



To edit parameters:

1. In the KULT Extension side bar, select the library.
2. Select the module.
3. If the KULT Module is not displayed, under Functions, select **Show Module View**.
4. To:
   - **Add a parameter:** Select **New**. Enter parameter values.
   - **Modify a parameter:** Change the parameter value in the fields.
   - **Delete a parameter:** Select a parameter, then select **Delete**.
     Refer to the following table for detail on the parameter values
5. Select **Apply** to add the changes to the code. Changes are displayed in the gray read-only code at the top of the module.

User module parameter values

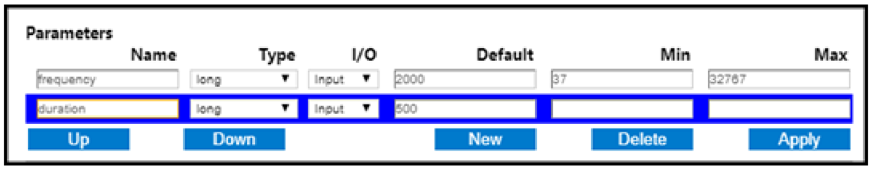
Name	Identifies the parameters that are passed to the user module.
Type	The parameter data type; only pointer types can be used for output parameters: ■ **char:** Character data ■ **char*:** Pointer to character data ■ **float:** Single-precision floating point data ■ **float*:** Pointer to single-precision floating point data ■ **double:** Double-precision data ■ **double*:** Pointer to double-precision point data ■ **int:** Integer data ■ **int*:** Pointer to integer data ■ **long:** 32-bit integer data ■ **long*:** Pointer to 32-bit integer data ■ **F_ARRAY_T:** Floating point array type ■ **I_ARRAY_T:** Integer array type ■ **D_ARRAY_T:** Double-precision array type
I/O	Defines whether the parameter is an input or output type.
Default	The default value for a nonarray (only) input parameter
Min	The minimum recommended value for a nonarray (only) input parameter. When the user module is used in a Clarius user test module (UTM), configuration of the UTM with a parameter value smaller than the minimum value causes Clarius to display an out-of-range message.
Max	The maximum recommended value for a nonarray (only) input parameter. When the user module is used in a Clarius UTM, configuration of the UTM with a parameter value larger than the maximum value causes Clarius to display an out-of-range message.

Reorder the user module parameters

To change the order of the parameters:

1. In the KULT Extension side bar, select the library.
2. Select the module.
3. In KULT Module, select the parameter.

Figure 90: Reorder parameters



Parameters

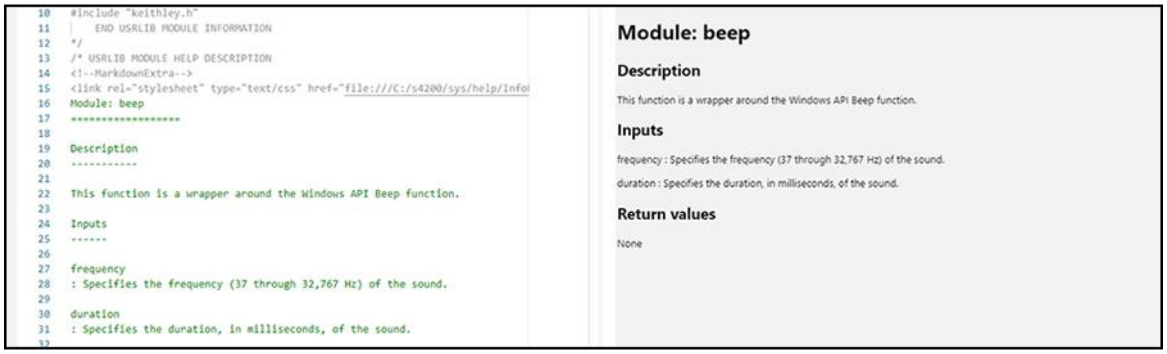
Name	Type	I/O	Default	Min	Max
frequency	long	Input	2000	37	32767
duration	long	Input	500		

4. Select **Up** or **Down** to move the parameter to the new location.
5. Select **Apply**. The changes are shown in the read-only code at the top of the editor.

Editing the module description

The module description appears in the help pane in Clarius when the module is selected. You can see a sample view of these descriptions in the KULT module in Visual Studio Code, beneath the parameters. This view is automatically updated when the description is edited.

Figure 91: Module description



```
#include "keithley.h"
    END USRLIB MODULE INFORMATION
*/
/* USRLIB MODULE HELP DESCRIPTION
<!--MarkdownExtra-->
<link rel="stylesheet" type="text/css" href="file:///C:/s4200/sys/help/Info
Module: beep
=================

Description
-----------

This function is a wrapper around the Windows API Beep function.

Inputs
------

frequency
: Specifies the frequency (37 through 32,767 Hz) of the sound.

duration
: Specifies the duration, in milliseconds, of the sound.
```

Module: beep

Description

This function is a wrapper around the Windows API Beep function.

Inputs

frequency : Specifies the frequency (37 through 32,767 Hz) of the sound.

duration : Specifies the duration, in milliseconds, of the sound.

Return values

None

NOTE

For a list of supported commands, refer to *Model 4200A-SCS LPT Library Programming*.

To edit the module description:

1. In the KULT side bar, select the library.
2. Select the module. The module is displayed in the editor.
3. Edit the description code below the read-only gray code at the top of the module, inside the comments for **USRLIB MODULE HELP DESCRIPTION**. The code uses Markdown syntax. For more information, see markdownguide.org.

Debugging libraries

In Visual Studio Code, you can attach a debugger to an execution process to monitor code execution for debugging purposes. For Keithley User Libraries, Visual Studio Code uses the GNU debugger (GDB) and attaches it to the UTMServer. Running the code as a UTM in Clarius allows the debugger to watch and control execution in the UTMServer.

To run and debug modules in Visual Studio Code, a launch configuration (`launch.json`) must exist in the `.vscode` folder. For instructions to set up a launch configuration, see Setting up Visual Studio Code for Library Development (on page 4-9).

Debug limitation notes:

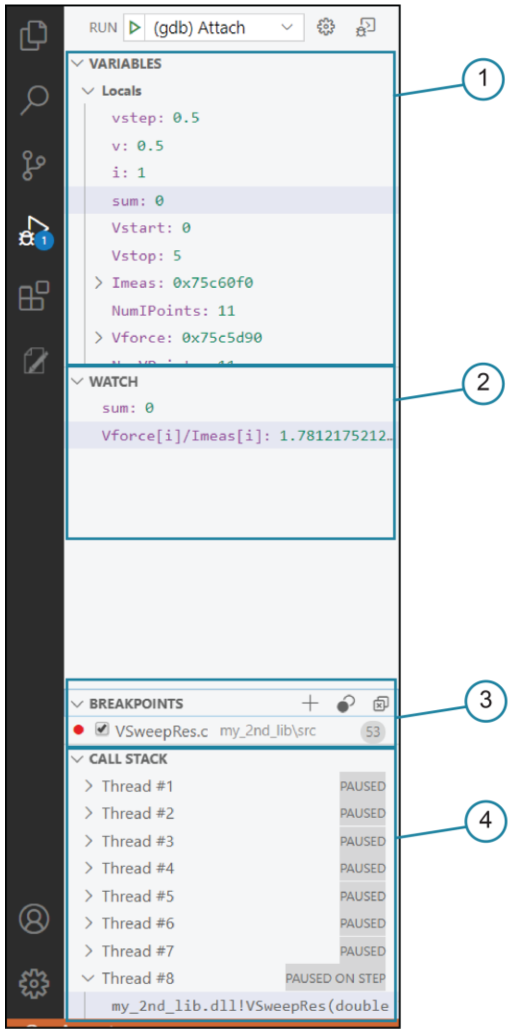
- While attached to the debugger, do not select Stop in Clarius. Selecting Stop may cause a Clarius process to hang. If this happens, open Windows Task Manager and select **End Task** for `Clarius.exe`, `KiteServer.exe`, and `UTMServer.exe`.
- The debugger is not available on the computer version of Clarius.

Debugger side bar and toolbar

The debugger allows you to step through code, monitor variables, evaluate expressions, and manipulate values. During debugging, the debug side bar and debug toolbar are visible.

The debug side bar gives you access to variable values, expressions, breakpoints, and threads for multi-threaded debugging.

Figure 92: Debugger side bar



```
vstep: 0.5
v: 0.5
i: 1
sum: 0
Vstart: 0
Vstop: 5
> Imeas: 0x75c60f0
NumIPoints: 11
> Vforce: 0x75c5d90
```

```
sum: 0
Vforce[i]/Imeas[i]: 1.7812175212…
```

```
VSweepRes.c  my_2nd_lib\src  53
```

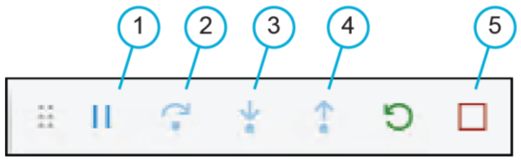
```
> Thread #1   PAUSED
> Thread #2   PAUSED
> Thread #3   PAUSED
> Thread #4   PAUSED
> Thread #5   PAUSED
> Thread #6   PAUSED
> Thread #7   PAUSED
v Thread #8   PAUSED ON STEP
  my_2nd_lib.dll!VSweepRes(double
```

Debugger side bar

1	Variables displays all variables. They are updated in real time as the code executes.
2	Watch allows you to add and monitor expressions and important variables.
3	Breakpoints allows you to add function breakpoints and manage other breakpoints.
4	Call Stack allows you watch the status of multiple threads.

The debugger toolbar gives users control over the code execution to step over or into lines of code.

Figure 93: Debugger toolbar



Debugger toolbar

1	Execute until the next breakpoint or the end of the code
2	Step over functions
3	Step into functions
4	Step out of functions
5	Detach

Setting up the debugger

This procedure builds the library for the debugger, which creates extra symbols that the debugger requires so that it can use breakpoints and watch variables.

NOTE

Libraries must be rebuilt after every change. Clarius can remain open but, the UTM must be reloaded for changes to take effect. Reload the UTM by select a different test and returning to the UTM, opening a different project and returning, or closing and reopening Clarius. Clarius must be fully closed to load a new library or module for the first time. You do not need to change to a different test if you are changing module content and not parameters of a module.

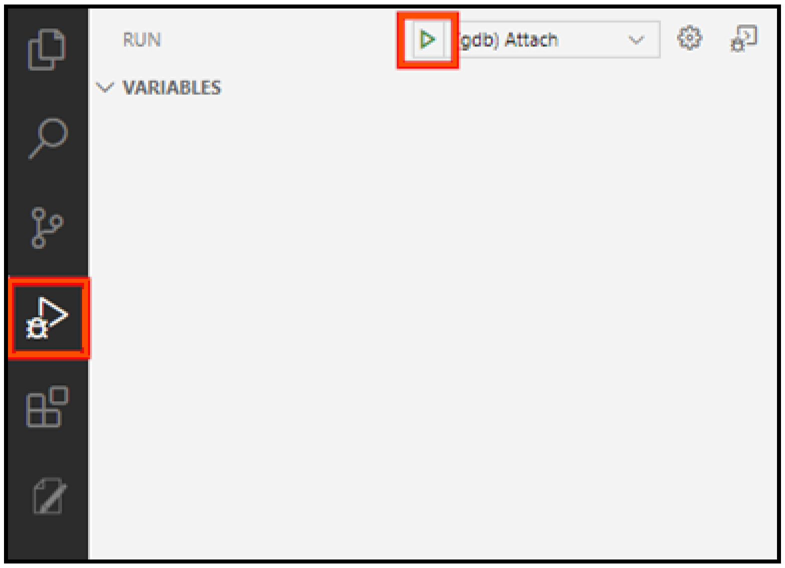
NOTE

When you start the debugger, breakpoints are unbound. The breakpoints automatically bind when code execution begins.

To set up the debugger for a module:

1. In the KULT side bar, select the library that contains the module.
2. Select the module.
3. Set at least one breakpoint. See Setting breakpoints in modules (on page 4-35).
4. Under Functions, run **Build Library LibName for Debug**.
5. Open Clarius.
6. In Clarius, either configure a new test to run the module or open an existing test that uses the module.
7. In Visual Studio Code, in the Debug side bar, select **Run > (gdb) Attach**. The debugger is running when the status bar at the bottom of the Visual Studio Code window is orange and debug toolbar is displayed.

Figure 94: Starting the debugger



Running code with the debugger

To run code with the debugger:

1. See the previous section to set up the debugger and the module in Clarius.
2. Run the UTM from Clarius.
3. When code execution is paused, you can use debugging tools, step through the code line by line, set additional breakpoints, or run to the next breakpoint using the debug toolbar.

Ending a debugging session

CAUTION

Do not abort a UTM in Clarius when execution is paused in Visual Studio Code. This causes a conflict between the debugger and Clarius and causes Clarius to hang. End the debugging session before aborting a UTM in Clarius.

To end a debugging session:

Select disconnect on the debug toolbar or type Ctrl+C into the terminal. This stops the running GNU Debugger (GDB) process.

If the code was paused on a breakpoint, it continues execution in Clarius after the debugger is disconnected. You can abort the module in Clarius after the debugger is disconnected.

Setting breakpoints in modules

Setting a breakpoint stops code execution and allows you to step through code line by line. Breakpoints must be set on lines of code. They do not work on comments or blank lines. You must set at least one breakpoint before attaching the debugger. You can set additional breakpoints during debugging when the code is paused. Breakpoints are marked as unbound (gray hollow circle) after starting until the code is executed.

The GDB environment allows the following breakpoints:

- **Unconditional breakpoint:** Pauses execution on a specific line
- **Conditional breakpoint:** Pauses execution on a specific line if a given statement is true
- **Function breakpoint:** Pauses execution at the first line of a function

Setting an unconditional breakpoint

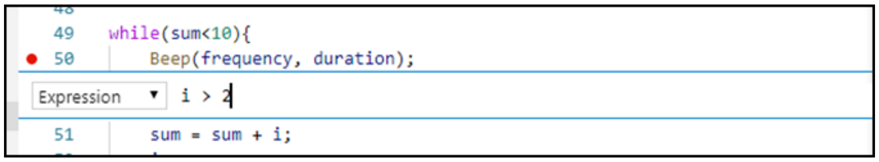
To set an unconditional breakpoint:

1. Set up the debugger as described in Setting up the debugger (on page 4-33).
2. Before running the debugger, select the space to the left of the line number. A red dot indicates the placed breakpoint. The breakpoint is also logged in the debug side bar under Breakpoints.
3. Under Functions, run **Build Library LibName for Debug**.
4. Open Clarius.
5. In Clarius, either configure a new test to run the module or open an existing test that uses the module.
6. In Visual Studio Code, in the Debug side bar, select **Run > (gdb) Attach**. The debugger is running when the status bar at the bottom of the Visual Studio Code window is orange and debug toolbar is displayed. When code execution is paused by a breakpoint, additional breakpoints can be added.

Setting a conditional breakpoint

To set a conditional breakpoint:

1. Before running the debugger, right-click the space to the left of the line number.
2. Select **Add Conditional Breakpoint**.
3. Type in an expression to be evaluated in the editor. This expression is evaluated before the line is executed and pauses execution if true.



```
while(sum<10){
    Beep(frequency, duration);
Expression ▼  i > 2
    sum = sum + i;
```

The breakpoint is also logged in the debugging side bar under Breakpoints.

4. Under Functions, run **Build Library LibName for Debug**.
5. Open Clarius.
6. In Clarius, either configure a new test to run the module or open an existing test that uses the module.
7. In Visual Studio Code, in the Debug side bar, select **Run > (gdb) Attach**. The debugger is running when the status bar at the bottom of the Visual Studio Code window is orange and debug toolbar is displayed. Once code execution is paused on a breakpoint, additional breakpoints can be added.

Setting a function breakpoint

To set a function breakpoint:

1. In the debug side bar under Breakpoints, select **+** to add a function breakpoint.
2. Type the name of the function. The breakpoint is verified the first time the code is executed and stops execution at the first line of the function.
3. Under Functions, run **Build Library LibName for Debug**.
4. Open Clarius.
5. In Clarius, either configure a new test to run the module or open an existing test that uses the module.
6. In Visual Studio Code, in the Debug side bar, select **Run > (gdb) Attach**. The debugger is running when the status bar at the bottom of the Visual Studio Code window is orange and debug toolbar is displayed. Once code execution is paused on a breakpoint, additional breakpoints can be added.

Expression evaluation

Visual Studio Code allows you to watch and evaluate expressions while code executes. You can also modify variables. The modifications to variables remain for the rest of the execution.

Evaluating an expression once

When you evaluate an expression once, the value of the expression is output to the debug Console.

To evaluate an expression once:

1. Set up the debugger as described in Setting up the debugger (on page 4-33).
2. Set up at least one breakpoint to pause the code.
3. Run the code in Clarius.
4. While the code is paused on a breakpoint, enter the expression to be evaluated into the Debug Console pane.

Evaluating an expression at every breakpoint

In this procedure the expression is evaluated every time the code is paused on a breakpoint. Additional expressions can be added at any time.

To evaluate an expression at every breakpoint:

1. Open the Debug side bar.
2. In the Debug side bar, select **+** in the Watch pane.
3. Enter the expression.
4. Set up the debugger as described in Setting up the debugger (on page 4-33).
5. Ensure at least one breakpoint is set to pause the code.
6. Run the code in Clarius.

Editing a variable value

Edited values are used for the remainder of the code execution.

To edit a variable value:

1. Set up the debugger as described in Setting up the debugger (on page 4-33).
2. Set up at least one breakpoint to pause the code.
3. Run the code in Clarius.
4. When the code is paused on a breakpoint, in the Watch pane, select **+**. You can also enter an expression in the Debug Console.
5. Enter an expression to change the value of a variable (`varName = newValue`).

Watching variables

All variables are visible in the Variables pane of the Debug side bar. You can watch specific variables by adding them to the Watch pane. Values are updated in real time.

To add a variable to the Watch pane:

1. Set up the debugger as indicated in Setting up the debugger (on page 4-33).
2. Set up at least on breakpoint to pause the code.
3. Run the code in Clarius.
4. When the code is paused on a breakpoint, right-click the variable in the Variables pane and select **Add to Watch**.

Section 5

KULT Extension tutorials

In this section:

Tutorial overview

The KULT Extension is a tool for Visual Studio Code that helps you develop user libraries. Each user library is comprised of one or more user modules. Each user module is created using the C programming language.

The following tutorials provide step-by-step instructions for creating user libraries and user modules in the KULT Extension.

The tutorials include:

- Creating a new user library and user module (on page 5-2): This tutorial shows you how to create a new user library and a new user module using the KULT Extension in Visual Basic Code. A hands-on example is provided that shows you how to create a user library that contains a user module that activates the internal beeper of the 4200A-SCS. You then build and run the module in Clarius. This tutorial also explores some of the features of Visual Studio Code to assist with writing code. This tutorial assumes a working knowledge of the C programming language.
- Creating a user module that returns data arrays (on page 5-13): This tutorial demonstrates the use of array variables in the KULT Extension. It also illustrates the use of return types (or codes), and the use of two functions from the Keithley Linear Parametric Test Library (LPTLib).
- Calling one user module from another (on page 5-20): This tutorial demonstrates how to set up user modules to call other user modules from any user library. It also describes how to copy a module.
- Customizing a user test module (UTM) (on page 5-23): This tutorial demonstrates how to modify a user module using the KULT Extension.

- Debugging a user module (on page 5-28): This tutorial demonstrates how to use the KULT Extension in Visual Studio Code to debug code with the GNU Debugger (GDB). The tutorial shows you how to pause execution, monitor variables and expressions, and step through code one line at a time.

Tutorial: Creating a new user library and user module

This tutorial shows you how to create a new user library and a new user module using the KULT Extension in Visual Basic Code. A hands-on example is provided that shows you how to create a user library that contains a user module that activates the internal beeper of the 4200A-SCS. You then build and run the module in Clarius. This tutorial also explores some of the features of Visual Studio Code to assist with writing code. This tutorial assumes a working knowledge of the C programming language.

This tutorial does not generate data. For an example of a user module that returns data, see Tutorial: Creating a user module that returns data arrays (on page 5-13).

Starting Visual Studio Code

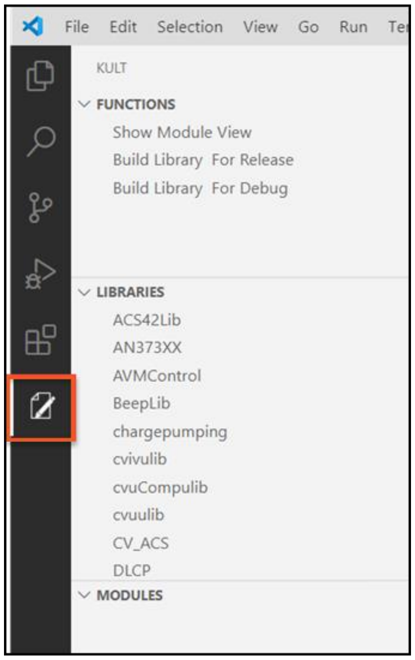
NOTE

Complete Installation (on page 4-1) before using this tutorial.

To start Visual Studio Code:

1. In the Windows Start menu, select Visual Studio Code.
2. Select the KULT icon to open the KULT side bar.

Figure 95: Opening the KULT side bar in Visual Studio Code

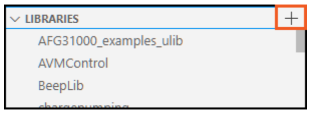


Creating a new user library

To create a new user library:

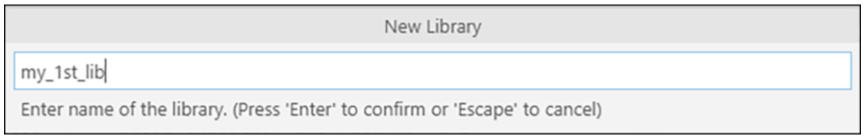
1. In the KULT side bar, in Libraries, select **+**.

Figure 96: Add a new user library



2. Enter `my_1st_lib` as the new user library name.

Figure 97: New library name


New Library
my_1st_lib
Enter name of the library. (Press 'Enter' to confirm or 'Escape' to cancel)

3. Select **Enter**. The library is displayed in the list of libraries. The necessary build files are automatically created.

Creating a new user module

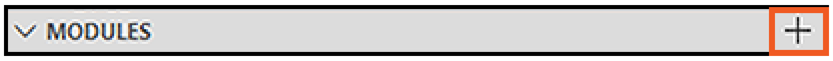
The names for user modules must:

- Conform to case-sensitive C programming language naming conventions.
- Be unique. They cannot duplicate names of existing user modules or user libraries.

To create a new user module:

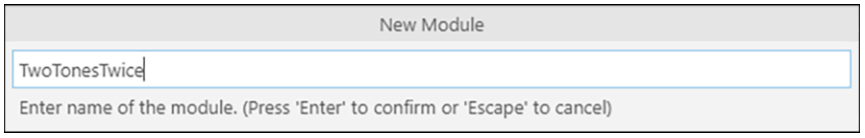
1. In the KULT side bar, from Libraries, select `my_1st_lib`. Under Modules, the modules in the library are displayed.
2. Select **Modules**.
3. Select **+** to add a module to the selected library.

Figure 98: Create a new user module



4. Enter `TwoTonesTwice` as the new user module name.

Figure 99: Naming a new module


New Module

TwoTonesTwice

Enter name of the module. (Press 'Enter' to confirm or 'Escape' to cancel)

5. Press **Enter** to apply the name. The module is displayed in the list of modules for the library.
6. Select the module in the KULT side bar. It is displayed in the Editor.

Entering a return type

If your user module generates a return value, you would select the data type from the Return Type list.

The `TwoTonesTwice` module does not produce a return value, so leave the Return Type at `void`.

Entering user module code

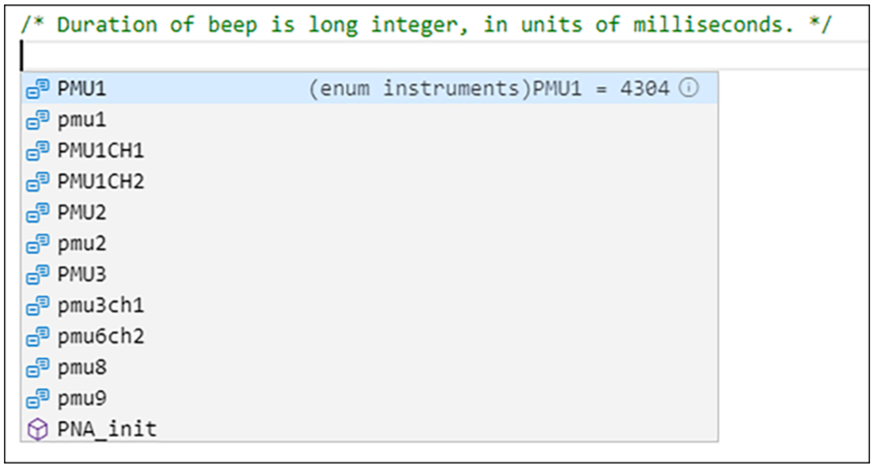
To add code to the module:

1. Enter the following comments that describe the purpose of the user module between comment lines `USRLIB MODULE CODE` and `USRLIB MODULE END`.

   ```
   /* Beeps four times at two alternating user-settable frequencies. */
   /* Makes use of Windows Beep (frequency, duration) function. */
   /* Frequency of beep is long integer, in units of Hz. */
   /* Duration of beep is long integer, in units of milliseconds. */
   ```

2. On the next line, press **Ctrl+Space** to open a list of all code suggestions.

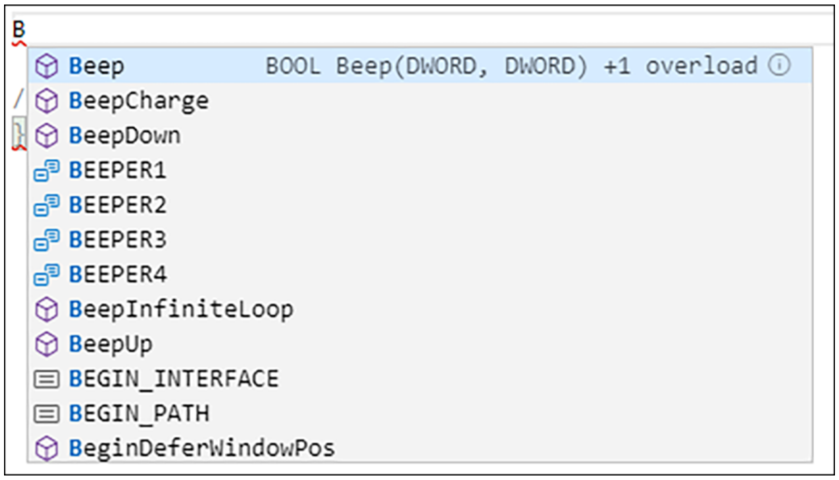
Figure 100: Add code to the module



```
/* Duration of beep is long integer, in units of milliseconds. */
|
PMU1                    (enum instruments)PMU1 = 4304
pmu1
PMU1CH1
PMU1CH2
PMU2
pmu2
PMU3
pmu3ch1
pmu6ch2
pmu8
pmu9
PNA_init
```

3. Type `Beep` to filter the suggestions. The list filters as you type.

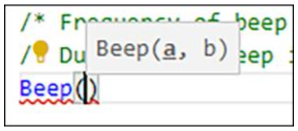
Figure 101: Add code to the module - filtered list



```
B
Beep                BOOL Beep(DWORD, DWORD) +1 overload
BeepCharge
BeepDown
BEEPER1
BEEPER2
BEEPER3
BEEPER4
BeepInfiniteLoop
BeepUp
BEGIN_INTERFACE
BEGIN_PATH
BeginDeferWindowPos
```

4. Select `Beep`. The function name is filled in automatically.
5. Continue the line by typing. A function prototype model is displayed. The bold underlined parameter is the next parameter to be entered.

Figure 102: Entering line of code



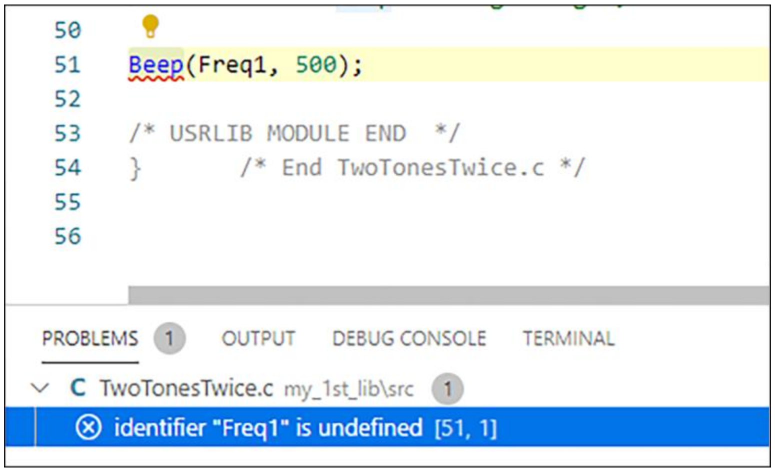
```
/* Frequency of beep
/  Du Beep(a, b) eep
Beep()
```

6. For `a`, type the parameter value `Freq1`.
7. For `b`, type the parameter value `500`.
8. End the function with a closing parenthesis and a semicolon.
9. Add the comment shown below:

```
Beep(Freq1, 500); /* Beep at first frequency for 500 ms */
```

10. Note that there is now a problem in the Problems tab at the bottom. Open the tab and select the problem.
11. The new line of code is highlighted and an indicator of the problem is displayed. In this case, the parameter `Freq1` is undefined. This is because `Freq1` was not added as a parameter yet. This will be defined later in the tutorial.

Figure 103: Identifier

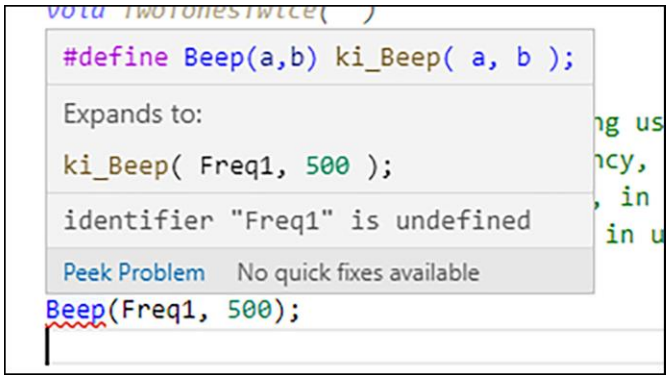


```
Beep(Freq1, 500);

/* USRLIB MODULE END  */
}       /* End TwoTonesTwice.c */
```

```
identifier "Freq1" is undefined [51, 1]
```

12. Hold the cursor over the new Beep function. Note that it expands to show details about the function.

Figure 104: Function description details



```
#define Beep(a,b) ki_Beep( a, b );
Expands to:
ki_Beep( Freq1, 500 );
identifier "Freq1" is undefined
```

```
Beep(Freq1, 500);
```

13. Enter the C code below. Note that the code deliberately contains a missing ; error to demonstrate a build error.

```
Beep(Freq2, 500); /* Beep at second frequency */
Beep(Freq1, 500);
Beep(Freq2, 500);
Sleep(500) /* NOTE deliberately forget semicolon */
```

Entering parameters

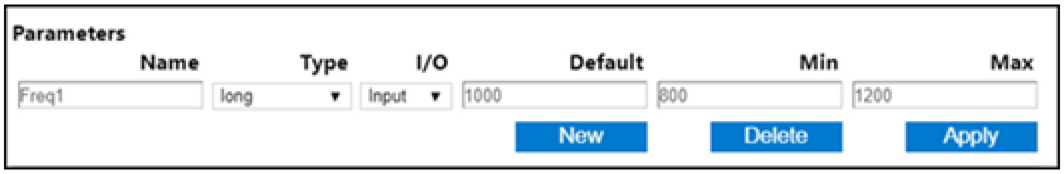
One of the parameters you enter is the data type; only pointer and array types can be used for output parameters. The available data types are:

- **char:** Character data
- **char*:** Pointer to character data
- **float:** Single-precision floating point data
- **float*:** Pointer to single-precision floating point data
- **double:** Double-precision data
- **double*:** Pointer to double-precision point data
- **int:** Integer data
- **int*:** Pointer to integer data
- **long:** 32-bit integer data
- **long*:** Pointer to 32-bit integer data
- **F_ARRAY_T:** Floating point array type
- **I_ARRAY_T:** Integer array type
- **D_ARRAY_T:** Double-precision array type

To enter the required parameters for the TwoTonesTwice user module:

1. In the KULT module, select **New**.
2. In the parameter name, enter `Freq1`.
3. For Type, select **long**.
4. For I/O, select **input**.
5. For Default, enter **1000**.
6. For Min, enter **800**.
7. For Max, enter **1200**.

Figure 105: Entering parameters

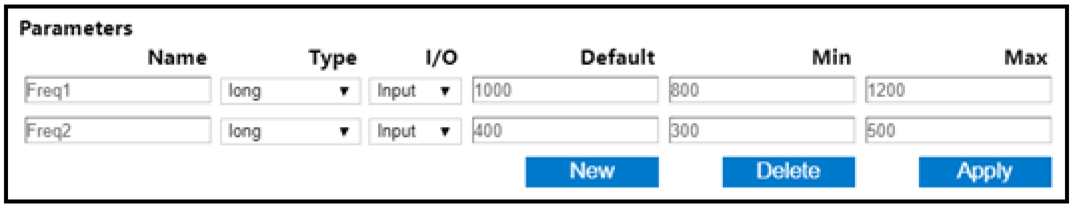


Parameters

Name	Type	I/O	Default	Min	Max
Freq1	long	Input	1000	800	1200

8. Add another parameter with the values:
   - **Parameter name:** `Freq2`
   - **Data type:** `long`
   - **I/O:** `Input`
   - **Default:** `400`
   - **Min:** `300`
   - **Max:** `500`

Figure 106: TwoTonesTwice parameters



Parameters

Name	Type	I/O	Default	Min	Max
Freq1	long	Input	1000	800	1200
Freq2	long	Input	400	300	500

9. Select **Apply** in the KULT Module. This adds the changes to the read-only code at the top of the module. Note that this removes errors from the Problems pane.

Entering header files

Any header files that are required are entered below the gray comment line `USRLIB MODULE PARAMETER LIST`. The header file `keithley.h` is added automatically when the module is created, since it is most commonly used. No additional header files are needed for this tutorial.

Documenting the user module

Module descriptions are entered between the comment lines `USRLIB MODULE HELP DESCRIPTION` and `END USRLIB MODULE HELP DESCRIPTION`. Code entered here in markdown format will appear in the Clarius help pane. To format the code, use Markdown, a web markup language. See markdownguide.org for information on using Markdown.

CAUTION

Do not use C-code comment designators (/*, */ or //) in the Description area. When the user module code is built, KULT evaluates the text in this area. C-code comment designators in the Description area can be misinterpreted, causing errors.

For the `TwoTonesTwice` user module, enter the following in the Description area:

```
<link rel="stylesheet" type="text/css"
href="http://clariusweb/HelpPane/stylesheet.css">

MODULE
======
TwoTonesTwice

DESCRIPTION
-----------
Execution results in sounding of four beeps at two alternating user-settable
frequencies. Each beeps sounds for 500 ms.

INPUTS
------
Freq1 (long) is the frequency, in Hz, of the first and third beep.
Freq2 (long) is the frequency, in Hz, of the second and fourth beep.

OUTPUTS
-------
None

RETURN VALUES
-------------
None
```

In the KULT module, the help information now appears below the parameters. The link at the top provides the Markdown style sheet used by the factory-provided module help panes and is not necessary for comments to be added to the help pane.

Saving the user module

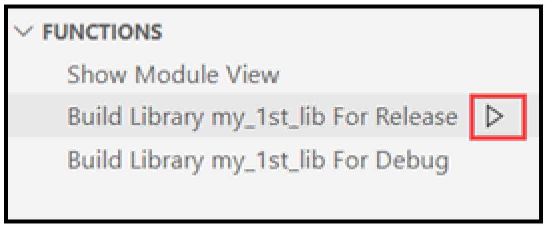
From the **File** menu, select **Save**.

Building the library

To build the library:

1. In the KULT side bar, under Function, select the run icon next to the function `Build Library my_1st_lib for Release`.

Figure 107: Building a library


FUNCTIONS
Show Module View
Build Library my_1st_lib For Release
Build Library my_1st_lib For Debug

2. In the Terminal tab at the bottom of the window, observe the build output. Note that it was unsuccessful.

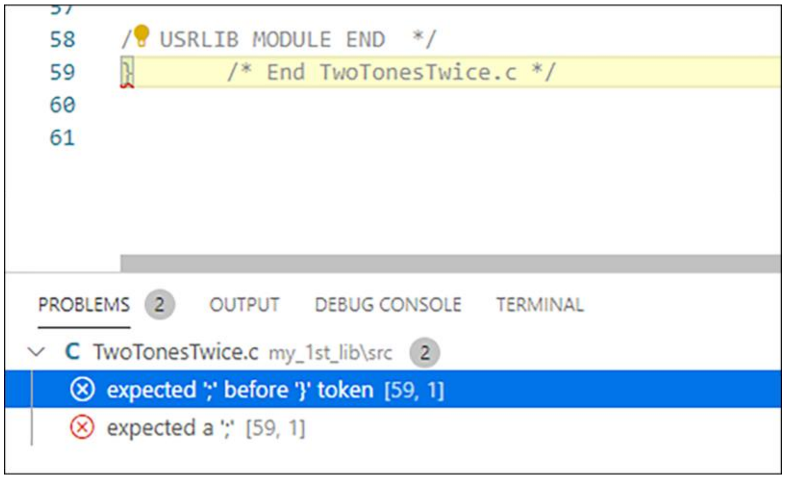
Finding code errors

In the Problems tab at the bottom of the window, you can review code errors. The error listing indicates the line with the error and a description of the problem.

To find code errors in the TwoTonesTwice user module:

1. Select the **Problems** tab at the bottom of the screen. There are two errors, one generated by the Intellisense feature and one generated by the build.
2. Select either of the problems. The line of code that caused the error is highlighted in the code editor.

Figure 108: Use the Problems tab to find code errors



```
/* USRLIB MODULE END  */
}        /* End TwoTonesTwice.c */
```

expected ';' before '}' token [59, 1]

expected a ';' [59, 1]

3. The error description indicates the problem. In this case, there is a missing semicolon before the closing brace. Correct the error by adding the missing semicolon (;).
4. Delete the error message. This also removes the Intellisense error. The build error is removed after a successful build.
5. Build the user library again.

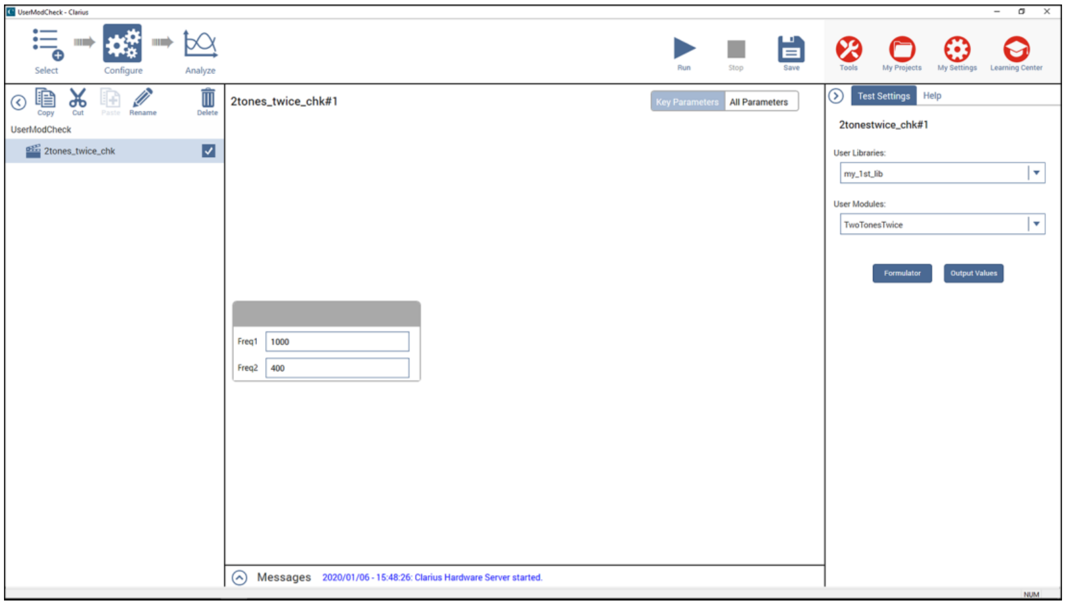
Checking the user module in Clarius

Check the user module in Clarius by setting up a user test module (UTM).

To check the module in Clarius:

1. Start Clarius.
2. Choose the **Select** pane.
3. Select **Projects**.
4. Select **New Project**.
5. Select **Create**. You are prompted to replace the existing project.
6. Select **Yes**.
7. Select **Rename**.
8. Enter `UserModCheck` and press **Enter**.
9. Select **Actions**.
10. Drag **Custom Action** to the project tree. The action has a red triangle next to it to indicate that it is not configured.
11. Select **Rename**.
12. Enter `2tones_twice_chk` and press **Enter**.
13. Select **Configure**.
14. In the Test Settings pane, select the `my_1st_lib` user library.
15. From the User Modules list, select the `TwoTonesTwice` user module. A group of parameters are displayed for the UTM as shown in the following figure. Accept the default parameters for now. You can experiment later after you establish that the user module executes correctly.

Figure 109: Configure the TwoTonesTwice UTM



Freq1 1000

Freq2 400

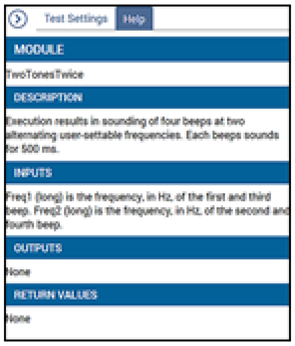
User Libraries: my_1st_lib

User Modules: TwoTonesTwice

2020/01/06 - 15:48:26: Clarius Hardware Server started.

16. Select **Help** to verify that the HTML in the Description tab is correctly formatted. An example is shown in the following figure.

Figure 110: Example Help



MODULE

TwoTonesTwice

DESCRIPTION

Execution results in sounding of four beeps at two alternating user-settable frequencies. Each beeps sounds for 500 ms.

INPUTS

Freq1 (long) is the frequency, in Hz, of the first and third beep. Freq2 (long) is the frequency, in Hz, of the second and fourth beep.

OUTPUTS

None

RETURN VALUES

None

17. Select **Save**.
18. Select **Run** to execute the UTM. You should hear a sequence of four tones, sounded at alternating frequencies.

Tutorial: Creating a user module that returns data arrays

This tutorial demonstrates the use of array variables in the KULT Extension. It also illustrates the use of return types (or codes), and the use of two functions from the Keithley Linear Parametric Test Library (LPTLib).

NOTE

Most of the basic steps that were detailed in Tutorial: Creating a new user library and user module (on page 5-2) are abbreviated in this tutorial.

Creating a new user library and user module

To name a new user library and new VSweep user module:

1. Open Visual Studio Code.
2. Open the KULT side bar.
3. Under Libraries, select **+** to create a new library.
4. Name the library `my_2nd_lib` and press **Enter**.
5. Select the library name.
6. Under Modules, select **+** to create a new user module in the library.
7. Name the module `VSweep` and press **Enter**.
8. Select the `VSweep` module to open it in the editor.

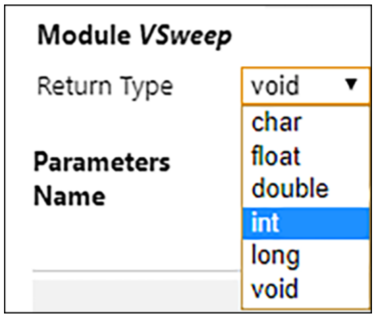
Entering the return type for VSweep

The VSweep user module generates an integer return value.

To set the return type of integer:

1. From the Return Type list, select **int**.
2. Select **Apply**.

Figure 111: VSweep Return Type setting



Entering the VSweep user module code

In the editor, enter the following C code for the VSweep user module between the comment lines `USRLIB MODULE CODE` and `USRLIB MODULE END`.

NOTE

When returning data using arrays, it is good practice to add a check to make sure that the points returned from a sweep is less than the size of the array. This prevents memory errors. This is not necessary here, since the array size is used as the number of points to calculate the step size. For modules that specify step size, the number of measurement points is always one greater than the number of steps.

```
double vstep, v; /* Declaration of module internal variables. */
int i;
if ( (Vstart == Vstop) ) /* Stops execution and returns -1 if */
return( -1 ); /* sweep range is zero. */
if ( (NumIPoints != NumVPoints) ) /* Stops execution and returns -2 if */
return( -2 ); /* V and I array sizes do not match. */
vstep = (Vstop-Vstart) / (NumVPoints -1); /* Calculates V-increment size. */
for(i=0, v = Vstart; i < NumIPoints; i++) /* Loops through specified number of */
/* data points. */
{
forcev(SMU1, v); /* LPTLib function forceX, which forces a V or I. */
measi(SMU1, &Imeas[i]); /* LPTLib function measX, which measures a V or I. */
/* Be sure to specify the *address* of the array. */
Vforce[i] = v; /* Returns Vforce array for display in UTM Sheet. */
v = v + vstep; /* Increments the forced voltage. */
}
return( 0 ); /* Returns zero if execution Ok.*/
```

Entering the VSweep user module parameters

This example uses the double-precision `D_ARRAY_T` array type. The `D_ARRAY_T`, `I_ARRAY_T`, and `F_ARRAY_T` are special array types that are unique to Keithley User Libraries. For each of these array types, you cannot enter values in the Default, Min, and Max fields. An extra parameter is created to indicate the array size.

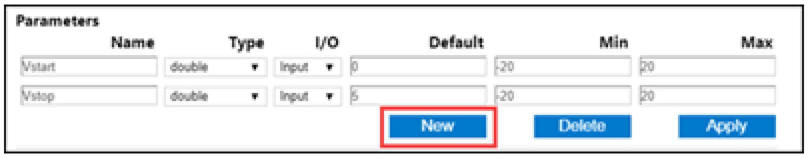
When executing the Vsweep user module in a Clarius UTM, the start and stop voltages (`Vstart` and `Vstop`) must differ. Otherwise, the first return statement in the code halts execution and returns an error number (-1). When a user module is executed using a Clarius UTM, this return code is stored in the UTM Data worksheet. The return code is stored in a column that is labeled with the user-module name.

When executing the VSweep user module in a Clarius UTM, the current and voltage array sizes must match; `NumIPoints` must equal `NumVPoints`. If the sizes do not match, the second return statement in the code halts execution and returns an error number (−2) in the VSweep column of the UTM Data worksheet.

To enter the required parameters:

1. In the KULT module, select **New** to create a new parameter.

Figure 112: Enter required code parameter



Parameters

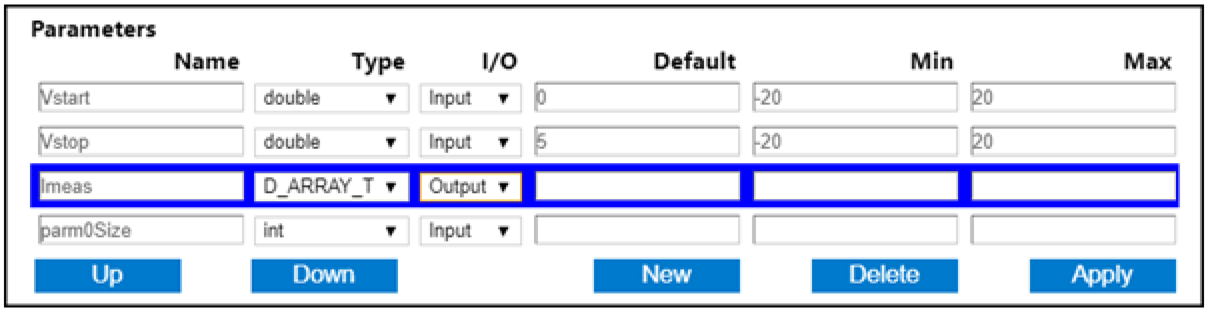
Name	Type	I/O	Default	Min	Max
Vstart	double	Input	0	-20	20
Vstop	double	Input	5	-20	20

2. Create the parameters `Vstart` and `Vstop` using the information in the following table.

Parameter Name	Type	I/O	Default	Min	Max
Vstart	double	Input	0	−20	20
Vstop	double	Input	5	−20	20

3. Select **New** to add a parameter for the measure current.
4. Enter the following parameter information:
   - Name: `Imeas`
   - Type: `D_ARRAY_T`
   - I/O: Output

Figure 113: KULT module parameters



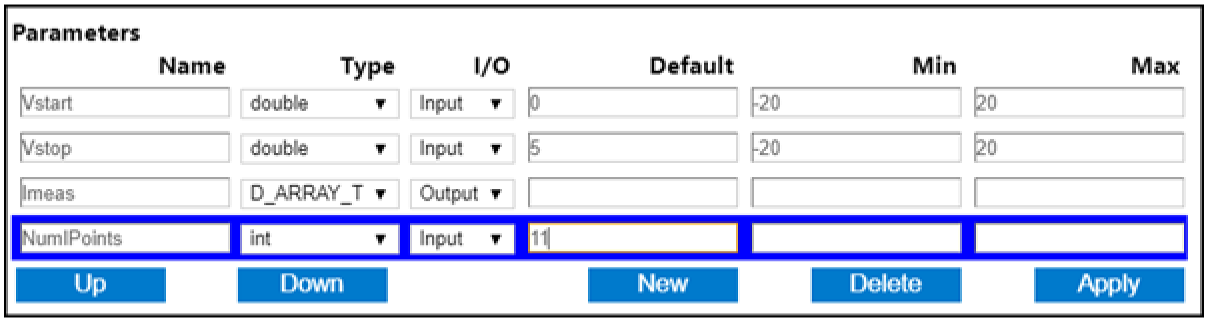
Parameters

Name	Type	I/O	Default	Min	Max
Vstart	double	Input	0	-20	20
Vstop	double	Input	5	-20	20
Imeas	D_ARRAY_T	Output			
parm0Size	int	Input			

5. The array size variable `parm0Size` was automatically added. Change the name to `NumIPoints`.

6. For `NumIPoints`, set the Default to **11**. You can also add Min and Max sizes if needed.

Figure 114: Specify the NumIPoints parameters

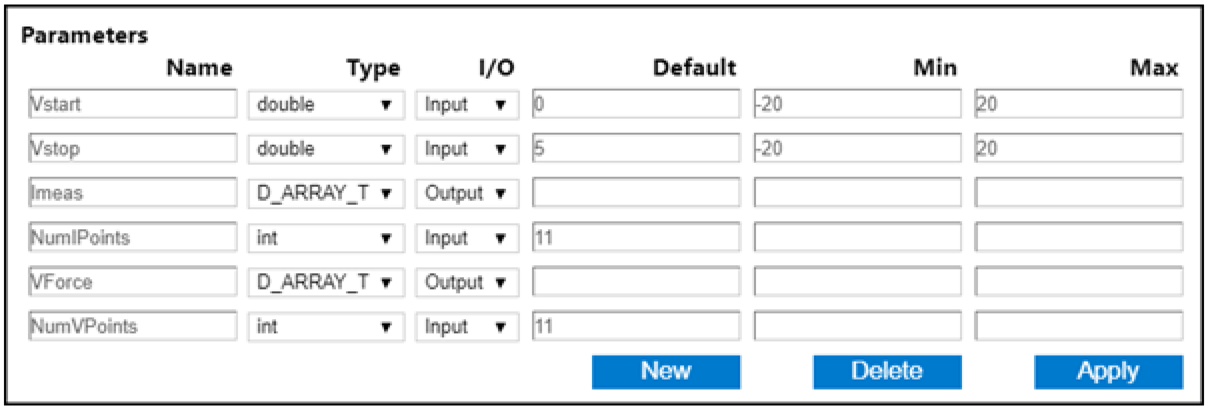


Parameters

Name	Type	I/O	Default	Min	Max
Vstart	double	Input	0	-20	20
Vstop	double	Input	5	-20	20
Imeas	D_ARRAY_T	Output			
NumIPoints	int	Input	11		

7. Select **New**.
8. Create a parameter for the forced voltage. Use the following settings:
    - Name: `Vforce`
    - Type: `D_ARRAY_T`
    - I/O: Output
9. Change the name of the automatically generated size parameter to `NumVPoints`.
10. For `NumVPoints`, set Default to **11**.
11. Select **Apply**. The user module contains the parameters shown in the following figure.

Figure 115: VSweep parameters



Parameters

Name	Type	I/O	Default	Min	Max
Vstart	double	Input	0	-20	20
Vstop	double	Input	5	-20	20
Imeas	D_ARRAY_T	Output			
NumIPoints	int	Input	11		
VForce	D_ARRAY_T	Output			
NumVPoints	int	Input	11		

Entering the header files for the VSweep user module

You do not need to enter any header files for the VSweep user module. The default `keithley.h` header file is sufficient.

Documenting the VSweep user module

Module descriptions are entered between the comment lines `USRLIB MODULE HELP DESCRIPTION` and `END USRLIB MODULE HELP DESCRIPTION`. Code entered here in markdown format will appear in the Clarius help pane. To format the code, use Markdown, a web markup language. See markdownguide.org for information on using Markdown.

A sample description is shown below:

```
<link rel="stylesheet" type="text/css"
   href="http://clariusweb/HelpPane/stylesheet.css">

VSweep module
--------------
Sweeps through a specified voltage range and measures current using a specified
   number of points.
Places forced voltage and measured current values (Vforce and Imeas) in output
   arrays.
NOTE For n increments, specify n+1 array size for both NumIPoints and NumVPoints.
```

Saving the VSweep user module

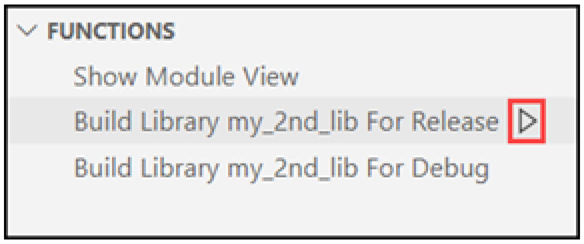
From the **File** menu, select **Save**.

Building the VSweep user module

To build the user module:

1. Under Functions, select **Build library my_2nd_lib for Release**.
2. Select the run icon.

Figure 116: Build the my_2nd_lib user library


FUNCTIONS
Show Module View
Build Library my_2nd_lib For Release
Build Library my_2nd_lib For Debug

3. Check the status of the build output in the Terminal tab at the bottom of the window.
4. Correct any errors and rebuild the user module.

Checking the VSweep user module in Clarius

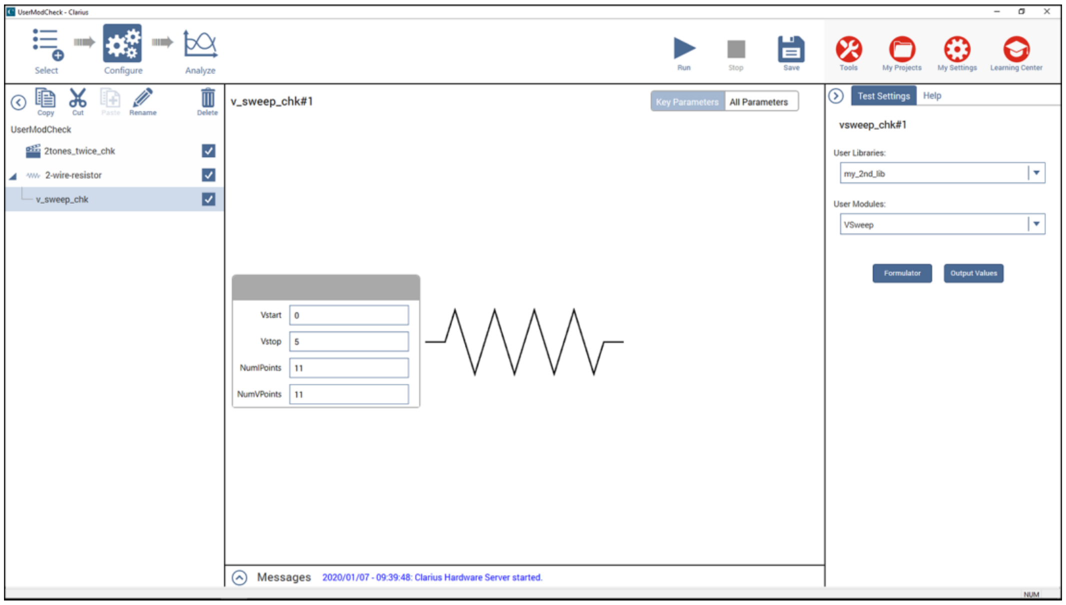
You can check the user module by adding it to a user test module (UTM) in Clarius and executing the UTM.

This procedure uses the project that was created for Tutorial: Creating a new user library and user module (on page 5-2).

To check the user module:

1. Connect a 1 kΩ resistor between the FORCE terminal of the ground unit (GNDU) and the FORCE terminal of SMU1.
2. Select the `UserModCheck` project.
3. Choose **Select**.
4. Select the **Devices** tab.
5. Select **Resistor, 2 terminal**.
6. Select **Add**.
7. Select the **Tests** tab.
8. Select **Custom Test**.
9. Select **Choose a test from the pre-programmed library (UTM)**.
10. Select **Add**. The test has a red triangle next to it to indicate that it is not configured.
11. Select **Rename**.
12. Enter the name `v_sweep_chk` and select **Enter**.
13. Select **Configure**.
14. In the right pane, from the User Libraries list, select the `my_2nd_lib` library.
15. From the User Modules list, select `VSweep`. A default schematic and group of parameters are displayed for the UTM.

Figure 117: Schematic and parameters for the v_sweep_chk UTM



v_sweep_chk#1

Vstart 0
Vstop 5
NumIPoints 11
NumVPoints 11

vsweep_chk#1

User Libraries: my_2nd_lib

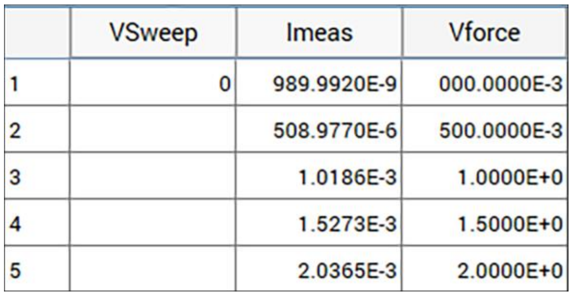
User Modules: VSweep

Messages 2020/01/07 - 09:39:48: Clarius Hardware Server started.

16. Select **Run**.
17. Select **Analyze**.
18. After execution, review the results in the Analyze sheet. The results should be similar to the results in the following figure. The current-to-voltage ratio for each row of results should be approximately 1 mA/V.

    In the first VSweep row, 0 is returned. This means that the user module executed without any errors.

Figure 118: Example of results from a UTM in the Analyze sheet



	VSweep	Imeas	Vforce
1	0	989.9920E-9	000.0000E-3
2		508.9770E-6	500.0000E-3
3		1.0186E-3	1.0000E+0
4		1.5273E-3	1.5000E+0
5		2.0365E-3	2.0000E+0

Tutorial: Calling one user module from another

This tutorial demonstrates how to set up user modules to call other user modules from any user library. It also describes how to copy a module.

In this tutorial, a new user module is created using the user modules created in the previous tutorials:

- Tutorial: Creating a new user library and user module (on page 5-2): The `TwoTonesTwice` user module, in the `my_1st_lib` user library, which is the independent user library that is called by the `VSweep` user module.
- Tutorial: Creating a user module that returns data arrays (on page 5-13): The `VSweep` user module in the `my_2nd_lib` user library, a copy of which is used as the dependent user library.

A copy of the `VSweep` user module, `VSweepBeep`, calls the `TwoTonesTwice` user module to signal the end of execution.

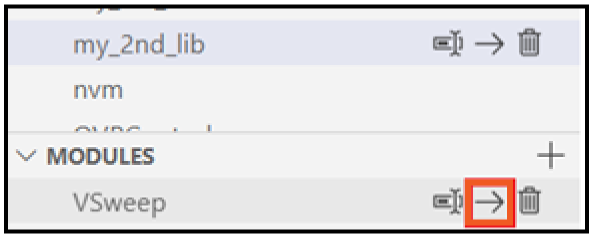
Copying an existing user module

In these steps, you copy the VSweep module to create the VSweepBeep module.

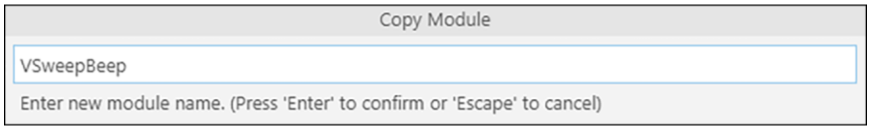
To copy the VSweep user module:

1. Start Visual Studio Code and open the KULT Extension.
2. From Libraries, select **my_2nd_lib**.
3. From Modules, select **VSweep**.
4. Select the copy icon next to the module to make a copy.

Figure 119: Copy the VSweep user module



5. Name the copied module **VSweepBeep**.


Copy Module

VSweepBeep

Enter new module name. (Press 'Enter' to confirm or 'Escape' to cancel)

6. Select **Enter**.
7. Select **VSweepBeep** in the side bar to open it in the editor.

Calling another user module from the VSweepBeep user module

To call the TwoTonesTwice user module at the end of the VSweepBeep user module:

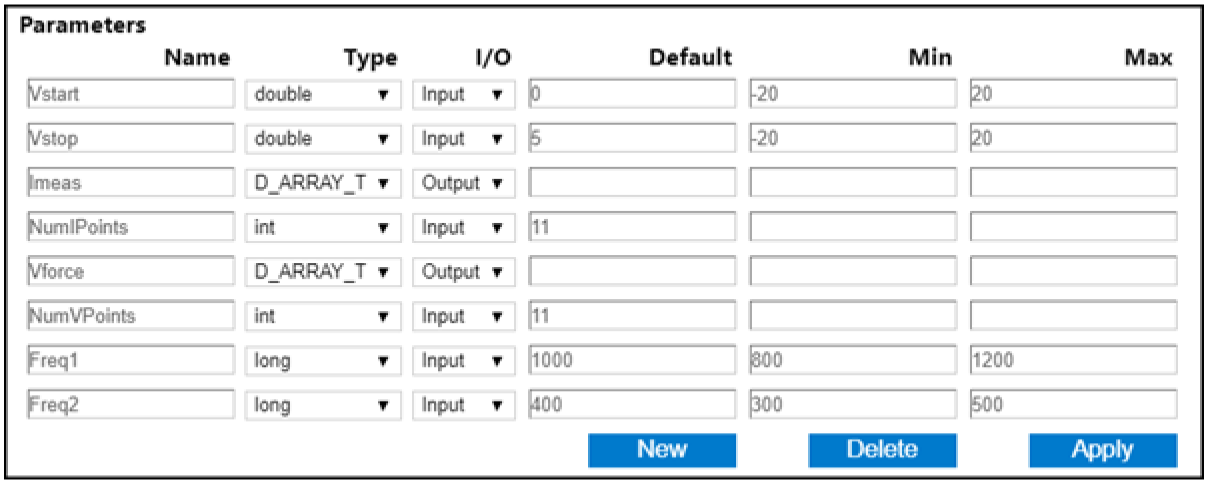
1. At the end of VSweepBeep, before the `return(0)` statement, add the following statement:

```
TwoTonesTwice(Freq1, Freq2); /* Beeps 4X at end of sweep. */
```

2. On the KULT module, add the `Freq1` and `Freq2` parameters shown in the following table and figure.

Name	Type	I/O	Default	Min	Max
Freq1	Long	Input	1000	800	1200
Freq2	Long	Input	400	300	500

Figure 120: VSweepBeep parameters



Parameters

Name	Type	I/O	Default	Min	Max
Vstart	double	Input	0	-20	20
Vstop	double	Input	5	-20	20
Imeas	D_ARRAY_T	Output			
NumIPoints	int	Input	11		
Vforce	D_ARRAY_T	Output			
NumVPoints	int	Input	11		
Freq1	long	Input	1000	800	1200
Freq2	long	Input	400	300	500

3. Select **Apply** to add the new parameters to the function prototype.

Specifying user library dependencies

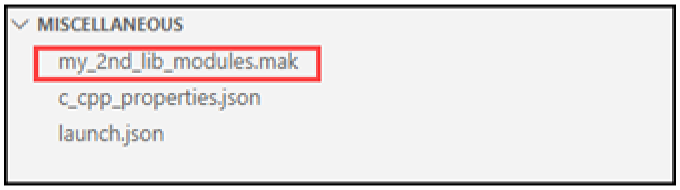
Before building the open user module, you must specify all the user libraries on which the user module depends.

The `VSweepBeep` user module depends on the `my_1st_lib` user library.

To specify the library dependency:

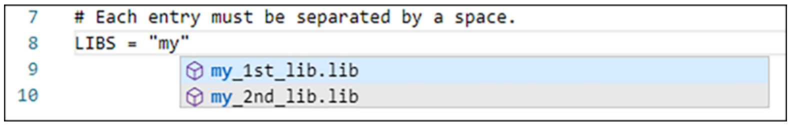
1. In the KULT side bar, under Miscellaneous, select `my_2nd_lib_modules.mak` to open it in the editor.

Figure 121: Select the .mak file



2. Place your cursor next to the `LIBS=` variable.
3. Press **Ctrl+Space** to display all libraries or type `my` to automatically filter.

Figure 122: Add library dependency



```
# Each entry must be separated by a space.
LIBS = "my"
          my_1st_lib.lib
          my_2nd_lib.lib
```

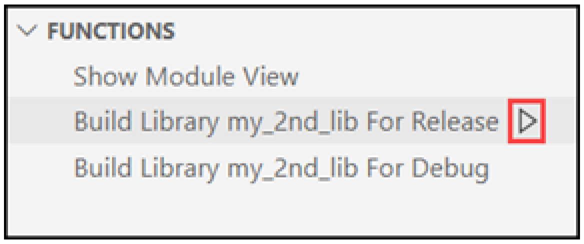
4. Select `my_1st_lib`.
5. Select **File > Save**.

Building the user library

To build the user library:

1. To save the `VSweepBeep` module, select **File > Save**.
2. Under Functions, select **Build library my_2nd_lib for Release**.
3. Select the run icon.

Figure 123: Build the my_2nd_lib user library


FUNCTIONS
Show Module View
Build Library my_2nd_lib For Release
Build Library my_2nd_lib For Debug

4. Check the build output for any errors.

Checking the VSweepBeep user module

Check the user module by creating and executing a user test module (UTM) in Clarius. Refer to Checking the user module. (on page 2-12)

This tutorial is almost identical to Tutorial: Creating a user module that returns data arrays (on page 5-13). The data produced should be the same as that tutorial. However, four beeps should sound at the end of execution.

Before proceeding:

1. Connect a 1 kΩ resistor between the FORCE terminal of the GNDU and the FORCE terminal of SMU1.
2. Instead of creating a new project, reuse the `UserModCheck` project that you created in Tutorial: Creating a new user library and user module (on page 5-2).
3. Add a UTM called `v_sweep_bp_chk`.
4. Configure the `v_sweep_bp_chk` UTM to execute the `VSweepBeep` user module, which is found in the `my_2nd_lib` user library.
5. Run the `v_sweep_bp_chk` UTM. Near the end of a successful execution, you should hear a sequence of four tones, sounded at alternating frequencies.
6. At the conclusion of execution, review the results in the Analyze sheet. If you connected a 1 kΩ resistor between SMU1 and GNDU, used the default UTM parameter values, and executed the UTM successfully, your results should be similar to the results shown in Checking the VSweep user module in Clarius (on page 5-18). The current/voltage ratio for each row of results should be approximately 1 mA/V.

Tutorial: Customizing a user test module (UTM)

This tutorial demonstrates how to modify a user module using the KULT Extension. In the `ivswitch` project, there is a test named `rdson`. The `rdson` test measures the drain-to-source resistance of a saturated N-channel MOSFET as follows:

1. Applies 2 V to the gate (Vg) to saturate the MOSFET.
2. Applies 3 V to the drain (Vd1) and performs a current measurement (Id1).
3. Applies 5 V to the drain (Vd2) and performs another current measurement (Id2).
4. Calculates the drain-to-source resistance rdson as follows:

```
rdson = (Vd2-Vd1) / (Id2-Id1)
```

The `rdson` test has a potential shortcoming. If the drain current is noisy, the two current measurements may not be representative of the actual drain current. Therefore, the calculated resistance may be incorrect.

In this example, the user module is modified in Visual Studio Code so that ten current measurements are made at Vd1 and ten more at Vd2. The current readings at Vd1 are averaged to yield Id1, and the current readings at Vd2 are averaged to yield Id2. Using averaged current readings smooths out the noise. The modified test, `rdsonAvg`, measures the drain-to-source resistance of a saturated MOSFET. The MOSFET is tested as follows when `rdsonAvg` is executed:

1. Applies 2 V to the gate (Vg) to saturate the MOSFET.
2. Applies 3 V to the drain (Vd1) and makes ten current measurements.
3. Averages the 10 current readings to yield a single reading (Id1).
4. Applies 5 V to the drain (Vd2) and makes ten more current measurements.
5. Averages the ten current readings to yield a single reading (Id2).
6. Calculates the drain-to-source resistance (`rdsonAvg`) as follows:

```
rdsonAvg = (Vd2-Vd1) / (Id2-Id1)
```

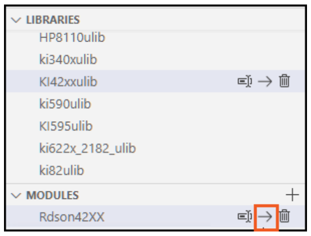
Copy the Rdson42XX user module

When naming a user module, conform to case-sensitive C programming language naming conventions. Do not duplicate names of existing user modules or user libraries.

To copy the user module:

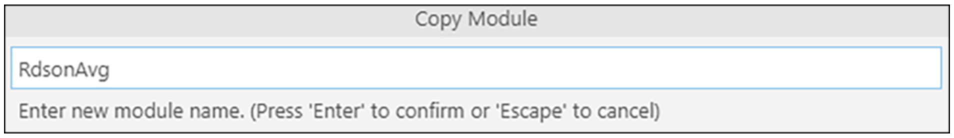
1. Open Visual Studio Code and the KULT side bar.
2. On the side bar under Libraries, select **KI42xxulib**.
3. Under Modules, select the **Rdson42XX** user module.
4. Select the copy icon.

Figure 124: Copy Rdson42XX module



5. Rename the copied module.

Figure 125: Name copied user module


Copy Module

RdsonAvg

Enter new module name. (Press 'Enter' to confirm or 'Escape' to cancel)

6. Press **Enter** to confirm the name.
7. Select the new module `RdsonAvg` to open it in the editor.

Modify the RdsonAvg user module

In the user module code, you need to replace the `measi` commands with `avgi` commands. While a `measi` command makes a single measurement, an `avgi` command makes a specified number of measurements, and then calculates the average reading. For example:

```
avgi(SMU2, Id1, 10, 0.01);
```

For the above command, SMU2 makes 10 current measurements and then calculates the average reading (`Id1`). The 0.01 parameter is the delay between measurements (10 ms).

The source code for the module is in the module code area of the window. In this area, make the following changes.

Under `Force the first point and measure`, change the line:

```
measi(SMU2, Id1);
```

to

```
avgi(SMU2, Id1, 10, 0.01); // Make averaged I measurement
```

Under `Force the second point and measure`, change the line:

```
measi(SMU2, Id2);
```

to

```
avgi(SMU2, Id2, 10, 0.01); // Make averaged I measurement
```

Change the line:

```
*Rdson = (Vd2-Vd1)/(*Id2- *Id1); // Calculate Rdson
```

to

```
*RdsonAverage = (Vd2-Vd1)/(*Id2- *Id1); // Calculate RdsonAverage
```

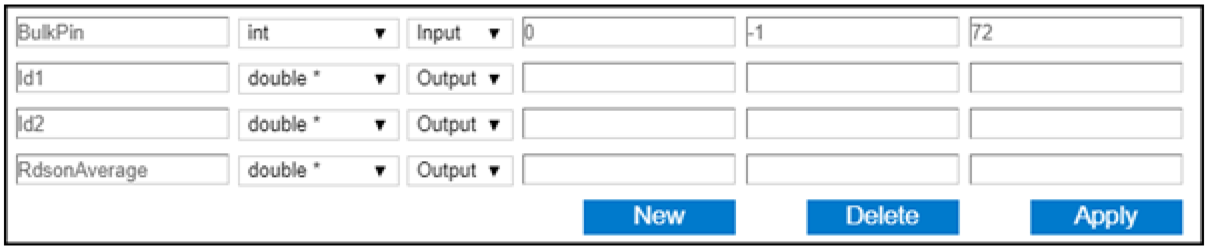
Change a parameter name

Parameters must have name that is different than the name of the user module.

To change the name of the Rdson parameter:

1. From the side bar, select **Show Module View**.
2. Select the name of the `Rdson` parameter.
3. Enter `RdsonAverage`.

Figure 126: Change the name of the Rdson parameter



BulkPin	int	Input	0	-1	72
Id1	double *	Output			
Id2	double *	Output			
RdsonAverage	double *	Output			

4. Select **Apply**.

Change the module description

In Clarius, any user test modules (UTMs) that are connected to this user module show the text that is entered in the Description section in the Clarius help pane.

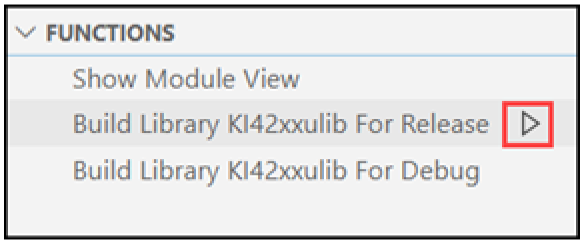
To change the module description:

1. Review the text between in the gray comments for `MODULE HELP DESCRIPTION`.
2. Replace all instances of `Rdson` with `RdsonAverage`.

Save and build the modified library

1. From the **File** menu, select **Save**.
2. Under Functions, select **Build library KI42xxulib for Release**.
3. Select the run icon.

Figure 127: Build the KI42xxulib library



FUNCTIONS
Show Module View
Build Library KI42xxulib For Release
Build Library KI42xxulib For Debug

4. Check the build output for errors.

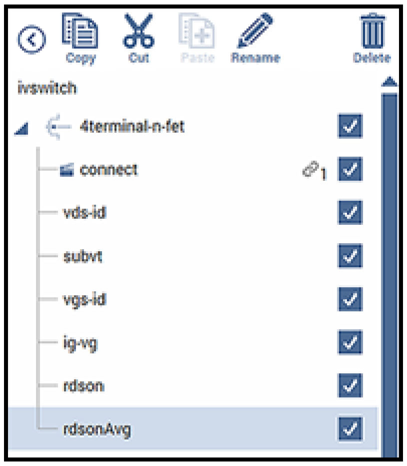
Add a new UTM to the ivswitch project

To add rdsonAvg to the ivswitch project:

1. Choose **Select**.
2. Select **Projects**.
3. In the Search box, enter **ivswitch** and select **Search**. The Library displays the I-V Switch Project (`ivswitch`).
4. Select **Create**. The `ivswitch` project replaces the previous project in the project tree.
5. Select the **Tests** tab.
6. For the Custom Test, select **Choose a test from the pre-programmed library (UTM)**.
7. Drag **Custom Test** to the project tree. The test has a red triangle next to it to indicate that it is not configured.
8. Select **Rename**.
9. Enter **rdsonAvg** and press **Enter**.
10. In the project tree, drag **rdsonAvg** to the `4terminal-n-fet device`, after the `rdson` test.
11. Choose **Configure**.
12. In the Test Settings pane, from the User Libraries list, select **KI42xxulib**.
13. From the User Modules list, select **Rdson42XX**.
14. Select **Save**.

The project tree for the `ivswitch` project with `rdsonAvg` added is shown in the following figure.

Figure 128: Add a new UTM to the ivswitch project



Tutorial: Debugging a user module

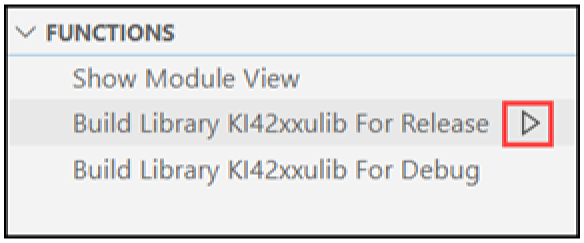
This tutorial demonstrates how to use the KULT Extension in Visual Studio Code to debug code with the GNU Debugger (GDB). The tutorial shows you how to pause execution, monitor variables and expressions, and step through code one line at a time.

Using copy to create the VSweepRes user module

To create the VSweepRes user module using copy:

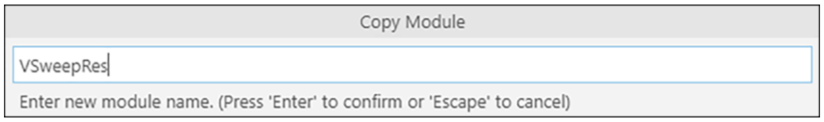
1. Start Visual Studio Code and open the KULT side bar.
2. From Libraries, select **my_2nd_lib**.
3. From Modules, select **VSweep**.
4. Select the copy icon.

Figure 129: Copy the VSweep user module


FUNCTIONS
Show Module View
Build Library KI42xxulib For Release
Build Library KI42xxulib For Debug

5. Name the copied module `VSweepRes`.

Figure 130: Name the copied module VSweepRes



VSweepRes
Enter new module name. (Press 'Enter' to confirm or 'Escape' to cancel)

6. Select **Enter**.
7. Select **VSweepRes** in the side bar to open it in the editor.

Adding an average resistance calculation to VSweepRes

To add a calculation for the average resistance:

1. At the beginning of `VSweepRes`, after the line defining `int i`, add the following statement:

   ```
   double sum = 0; /*Sum of all resistance measurements*/
   ```

2. Inside the for loop, after the line `v = v + vstep; /* Increments the forced voltage. */` add the following statement:

   ```
   sum =(Vforce[i]/Imeas[i]); /*Intentionally incorrect line*/
   ```

   - That line is intentionally incorrect. We will find the error using the debugger later.

3. After the for loop, before the return statement, add the following statement:

   ```
   *AvgRes = sum/(NumIPoints - 1); /*Divide by the number of measurements, not
   including 0 V, to get average. */
   ```

Adding a parameter to VSweepRes

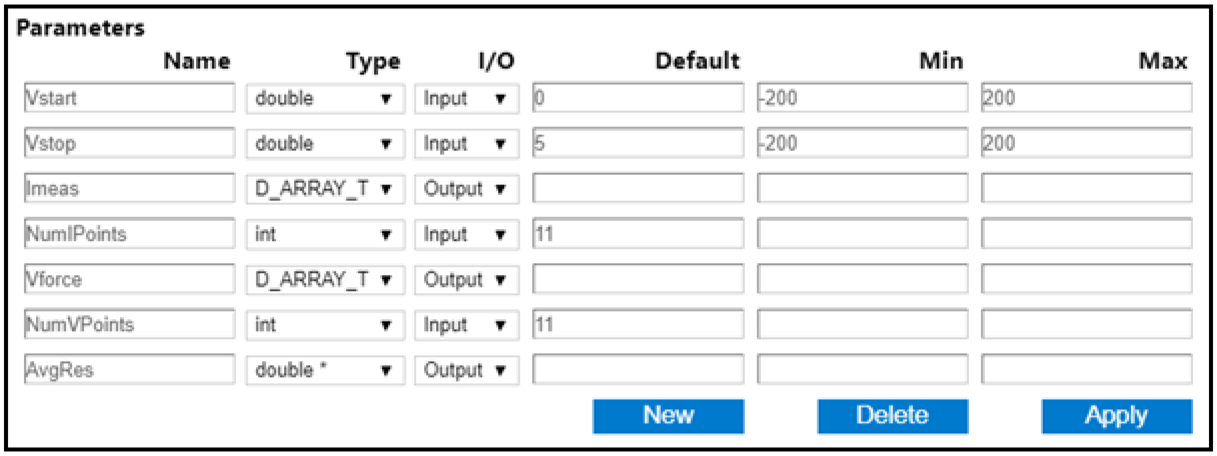
To add a parameter to VSweepRes:

1. On the KULT module, add a new parameter, `AvgRes`, with the values shown in the following table.

Name	Type	I/O	Default	Min	Max
AvgRes	Double*	Output			

2. Select **Apply** to add the new parameter.

Figure 131: Add a new parameter



Parameters

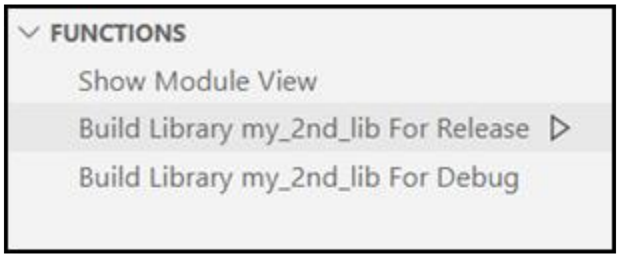
Name	Type	I/O	Default	Min	Max
Vstart	double	Input	0	-200	200
Vstop	double	Input	5	-200	200
Imeas	D_ARRAY_T	Output			
NumIPoints	int	Input	11		
Vforce	D_ARRAY_T	Output			
NumVPoints	int	Input	11		
AvgRes	double *	Output			

Building the user library

To build the user library:

1. From the **File** menu, select **Save**.
2. Under Functions, select **Build library my_2nd_lib for Release**.
3. Select the run icon.

Figure 132: Build my_2nd_lib library


FUNCTIONS
Show Module View
Build Library my_2nd_lib For Release
Build Library my_2nd_lib For Debug

4. Check the build output for any errors. The build should be successful.

Checking the VSweepRes user module

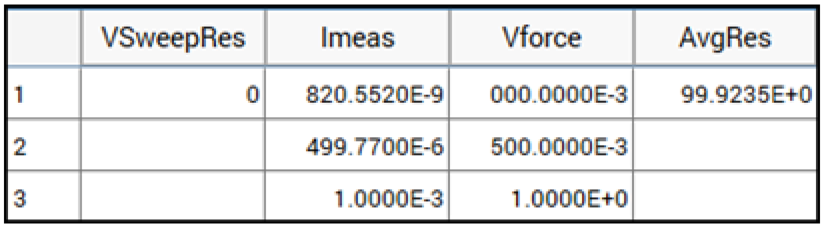
Check the user module by creating and executing a user test module (UTM) in Clarius.

To check the user module:

1. Connect a 1 kΩ resistor between the FORCE terminal of the ground unit (GNDU) and the FORCE terminal of SMU1.
2. Select the `UserModCheck` project.
3. Choose **Select**.
4. Select the **Devices** tab.
5. Select **Resistor, 2 terminal**.
6. Select **Add**.
7. Select the **Tests** tab.
8. Select **Custom Test**.
9. Select **Choose a test from the pre-programmed library (UTM)**.
10. Select **Add**. The test has a red triangle next to it to indicate that it is not configured.
11. Select **Rename**.
12. Enter the name `v_sweep_chk` and select **Enter**.
13. Select **Configure**.
14. In the right pane, from the User Libraries list, select the `my_2nd_lib` library.

15. From the User Modules list, select `VSweepRes`. A default schematic and group of parameters are displayed for the UTM.
16. Select **Run**.
17. Select **Analyze**.
18. Review the results in the Analyze sheet. The results should be similar to the results in the following figure. Notice that there is a new value returned in the sheet, `AvgRes`, which is the average calculated resistance. However, the value is incorrect. If you connected a 1 kΩ resistor, the value is closer to 100 Ω. There is something wrong in the user module. In the next topic, you use the debugger to help find the error.

Figure 133: Analyze results for v_sweep_res_chk



	VSweepRes	Imeas	Vforce	AvgRes
1	0	820.5520E-9	000.0000E-3	99.9235E+0
2		499.7700E-6	500.0000E-3	
3		1.0000E-3	1.0000E+0	

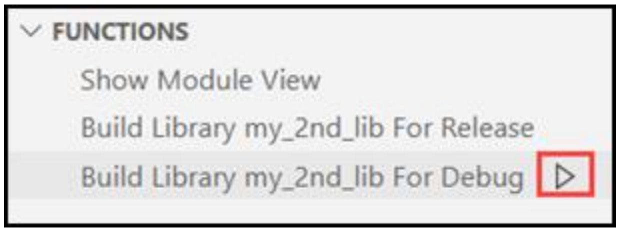
Starting the debugger and adding a breakpoint

At least one breakpoint must be set before running the debugger. Breakpoints bind when code execution begins.

To start the debugger and add a breakpoint:

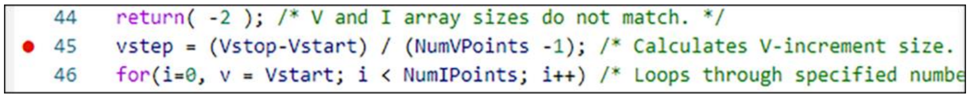
1. In Visual Studio Code, in the KULT side bar, select the library that contains the module.
2. Select the module.
3. Under Functions, select **Build Library my_2nd_lib for Debug**.
4. Select the run icon.

Figure 134: Build the library for debug


FUNCTIONS
Show Module View
Build Library my_2nd_lib For Release
Build Library my_2nd_lib For Debug

5. In Clarius, reload the user module by selecting another test, then selecting **v_sweep_res_chk** again.
6. Place an unconditional breakpoint by selecting the space to the left of the line that calculates the V-increment size. Code execution will pause at this line.

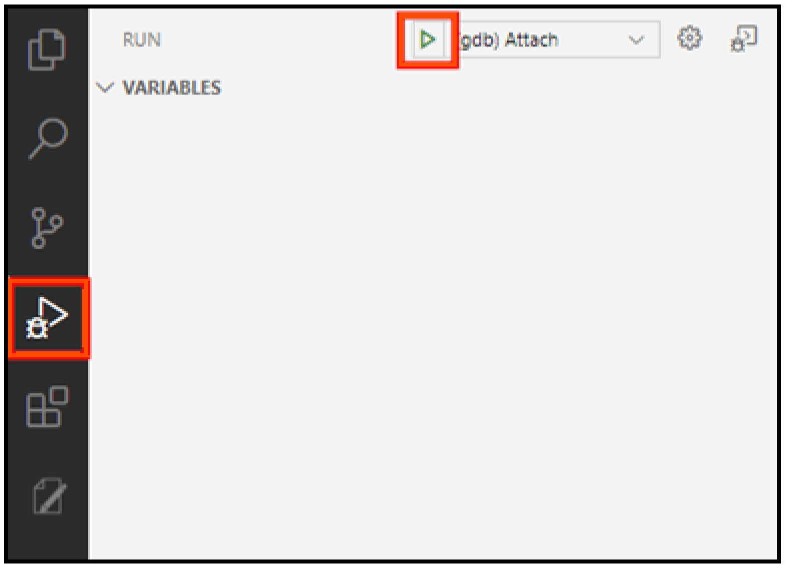
Figure 135: Unconditional breakpoint



```
return( -2 ); /* V and I array sizes do not match. */
vstep = (Vstop-Vstart) / (NumVPoints -1); /* Calculates V-increment size.
for(i=0, v = Vstart; i < NumIPoints; i++) /* Loops through specified numbe
```

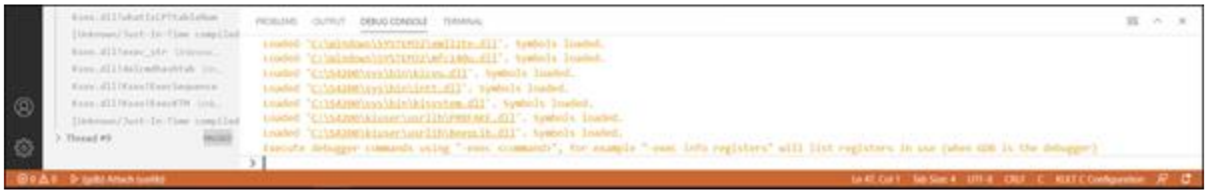
7. In Visual Studio Code, in the Debug side bar, select **Run > (gdb) Attach**.

Figure 136: Starting the debugger



8. Wait for the attach process to complete. The attach process is complete and the debugger is running when the status bar at the bottom changes from blue to orange as shown in the following figure.

Figure 137: Debug Console and status bar



Debugging the code

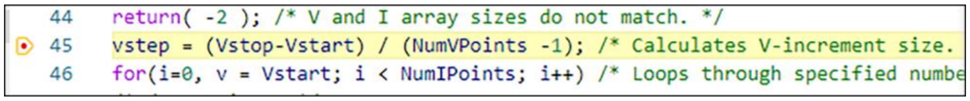
Once the attach process is complete, the code can be executed.

The attach process causes any previously set breakpoints to temporarily unbind (turn gray). They automatically rebind when code execution starts. You can change, add, or remove breakpoints when the code execution is paused on an existing breakpoint.

To debug the code:

1. In Clarius, select **Run** to start the code.
2. Return to Visual Studio Code. The code pauses on the breakpoint.

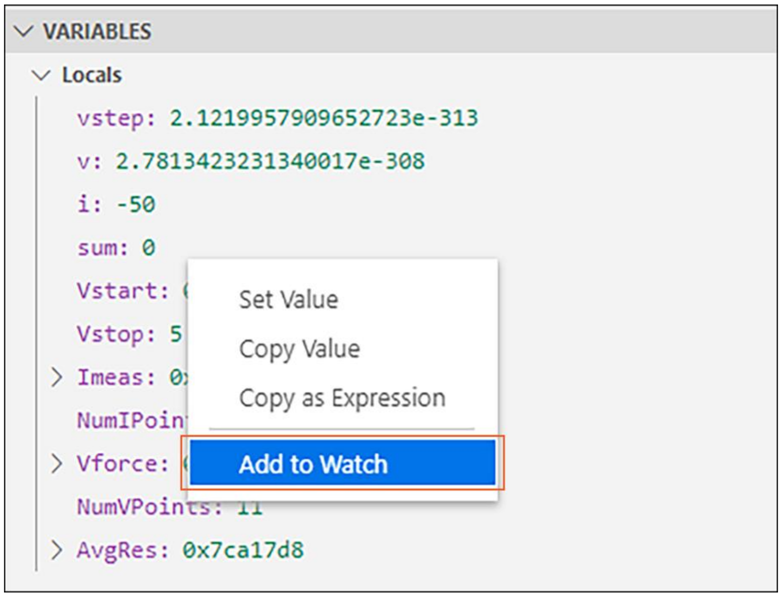
Figure 138: Code paused on breakpoint



```
return( -2 ); /* V and I array sizes do not match. */
vstep = (Vstop-Vstart) / (NumVPoints -1); /* Calculates V-increment size.
for(i=0, v = Vstart; i < NumIPoints; i++) /* Loops through specified numbe
```

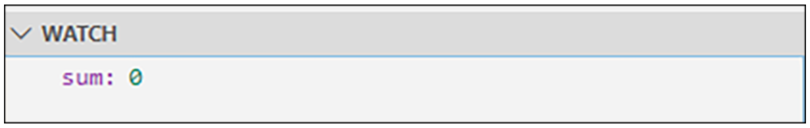
3. On the Variables pane in the debugger side bar, find the sum variable. Right-click the variable and select **Add to watch.** The variable is shown in the watch pane on the side bar.

Figure 139: Add to watch, variables pane



```
VARIABLES
  Locals
    vstep: 2.1219957909652723e-313
    v: 2.7813423231340017e-308
    i: -50
    sum: 0
    Vstart: (
    Vstop: 5
  > Imeas: 0x
    NumIPoin
  > Vforce: (
    NumVPoints: 11
  > AvgRes: 0x7ca17d8
```

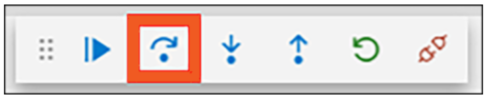
Figure 140: Variable sum added to watch



```
WATCH
  sum: 0
```

4. Select **Step Over** on the debugger toolbar until you get to the line `sum =(Vforce[i]/Imeas[i]);`. At that point, the code loops back to the top of the for loop. Continue until you get to the sum line again.

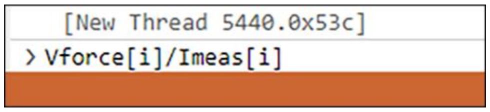
Figure 141: Step over



5. Select **Step Into**. This line has now executed and the `sum` value has changed.
6. Select **Debug Console** at the bottom of the screen. Enter the formula below:

```
Vforce[i]/Imeas[i]
```

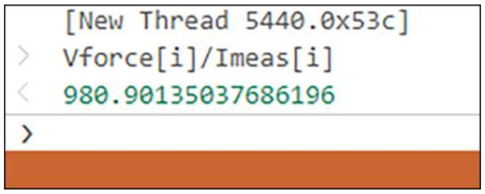
Figure 142: Enter formula in Debug Console



```
  [New Thread 5440.0x53c]
> Vforce[i]/Imeas[i]
```

7. Press **Enter**. The value returned is the same as the value in sum and is approximately the value of the resistor. This verifies that the correct resistance is calculated from the current measurement.

Figure 143: Returned value



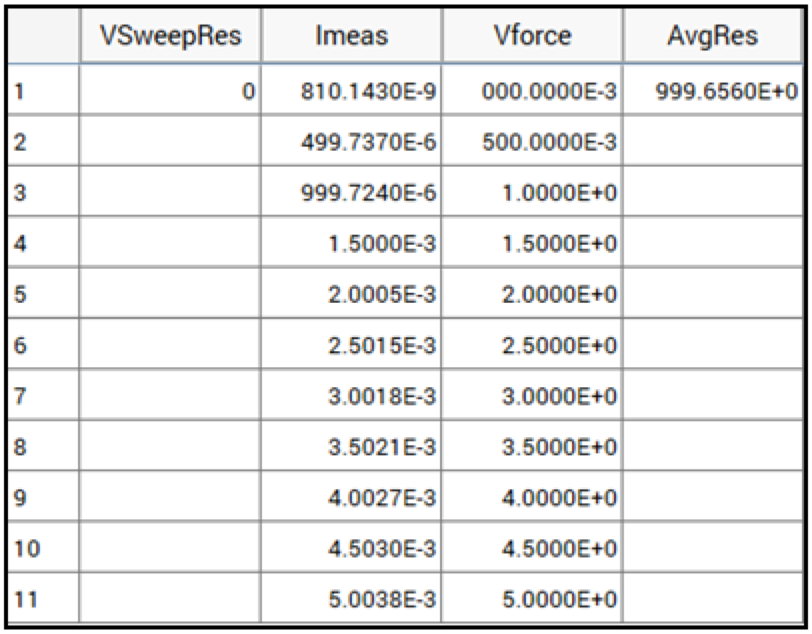
```
  [New Thread 5440.0x53c]
> Vforce[i]/Imeas[i]
< 980.90135037686196
>
```

8. Continue stepping through the code until you get to the top of the `for` loop again. The value for sum is not changing. Therefore, our sum formula must be incorrect.
9. Press **F5** the **Continue** button on the debug toolbar. This will run the code until completion.
10. Select to the **Terminal** tab at the bottom.
11. Select the **Disconnect** button. This terminates the debugging session.
12. Correct the sum line from:
    `sum =(Vforce[i]/Imeas[i]); /*Intentionally incorrect line*/`
    to
    `sum = sum + (Vforce[i]/Imeas[i]); /*Sum Resistances*/`
13. Rebuild the library for release by selecting the command **Build Library my_2nd_lib For Release** on the KULT side bar.

Retest the VSweepRes user module in Clarius

1. Return to Clarius. Click away from the `v_sweep_res_chk` test and back to reload the module.
2. Rerun the module with the same settings as before, 0 to 5 V with number of V and I points as 11.
3. Select the **Analyze** view. The resistance value is now correct.

Figure 144: Analyze the VSweepRes user module



	VSweepRes	Imeas	Vforce	AvgRes
1	0	810.1430E-9	000.0000E-3	999.6560E+0
2		499.7370E-6	500.0000E-3	
3		999.7240E-6	1.0000E+0	
4		1.5000E-3	1.5000E+0	
5		2.0005E-3	2.0000E+0	
6		2.5015E-3	2.5000E+0	
7		3.0018E-3	3.0000E+0	
8		3.5021E-3	3.5000E+0	
9		4.0027E-3	4.0000E+0	
10		4.5030E-3	4.5000E+0	
11		5.0038E-3	5.0000E+0	

Keithley Instruments • 28775 Aurora Road • Cleveland, Ohio 44139 • 1-800-833-9200 • tek.com/keithley

KEITHLEY
A Tektronix Company

04/2022