Using the Model 7701 Multiplexer in a DAQ6510 Switching-Only Application

Introduction

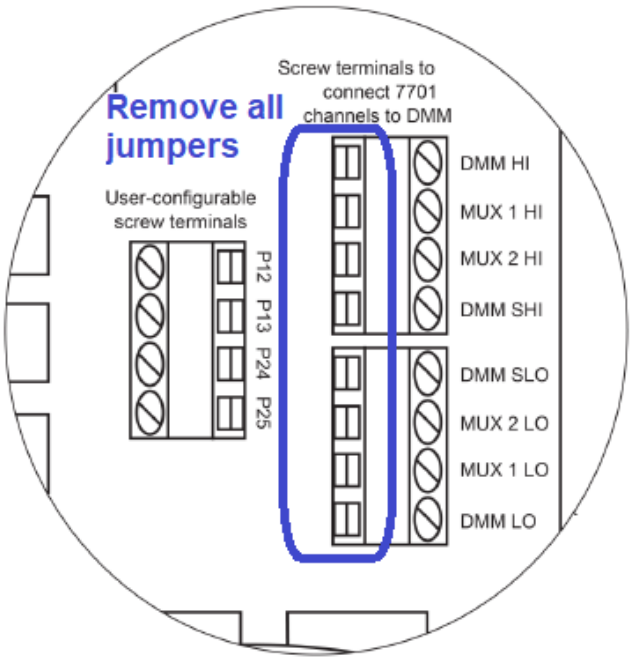
While the Keithley DAQ6510 datalogger integrates switching and measurement within the same mainframe, a user may have a need for additional flexibility and signal routing options. This may involve some specifically tailored switching patterns; the introduction of a stimulant source (like a power supply or a SourceMeter); or perhaps an alternate measurement tool instead of the internal 6.5-digit multimeter of the DAQ6510. While all the multiplexer module options provide this type of freedom, the Model 7701 offers an extra degree of control with respect to routing.

In this piece, we briefly explore some of the extended internal wiring aspects as well as how a user might use the DAQ6510 touchscreen controls for manually exercising all relays on the multiplexer module. We conclude with a remote programming example to highlight the few commands necessary to accomplish customized routing schemes.

Wiring Procedure

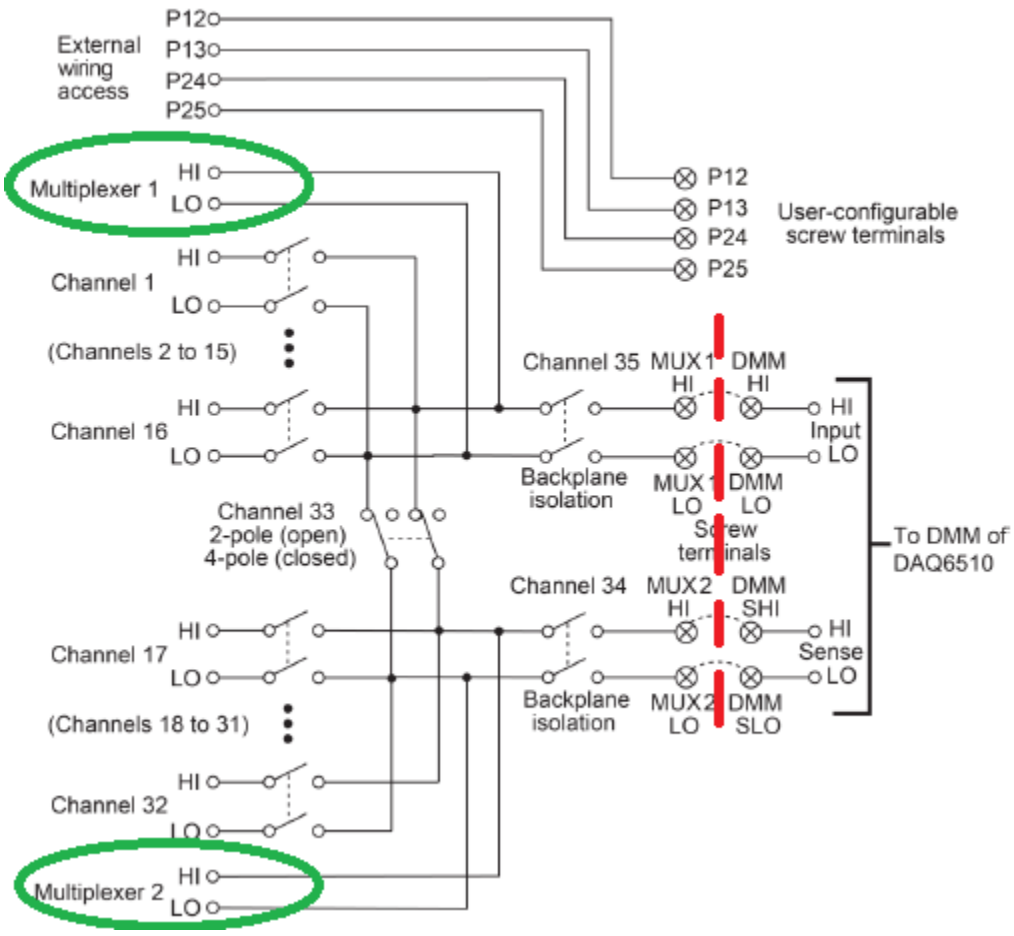
The Model 7701 provides the option of physically removing any possible internal connection to the DMM and SENSE inputs of the DAQ6510, while also allowing for the operator to introduce their own source signals through the MUX1 and MUX2 inputs. This is done by eliminating any/all jumpers at the terminal blocks inside the Model 7701 module enclosure.

Figure 2: 7701 screw terminals



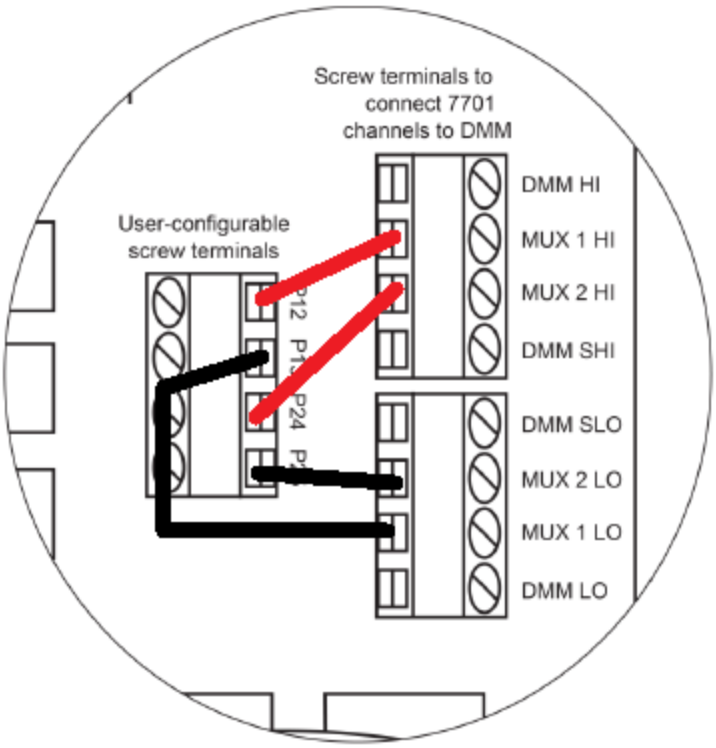
Notice how, in the simplified schematic below, this modification isolates the multiplexer (MUX) inputs from the DMM inputs.

Figure 10: 7701 simplified schematic



If the user wishes to expose additional signal paths to the outside of the instrument without consuming one of the available channels, there is the option of adding auxiliary signals to the MUX1 and MUX2 switch banks. This is done by adding jumpers such that the user configurable screw terminals (P12, P13, P24, and P25) are routed directly to the MUX1 and MUX2 HI/LO inputs. To do so, the user introduces jumpers at the terminal blocks inside the Model 7701 module enclosure. Below is an example of how P12 and P13 might pair up with the MUX1 HI/LO inputs and P24 and P25 might pair up with the MUX2 HI/LO inputs.

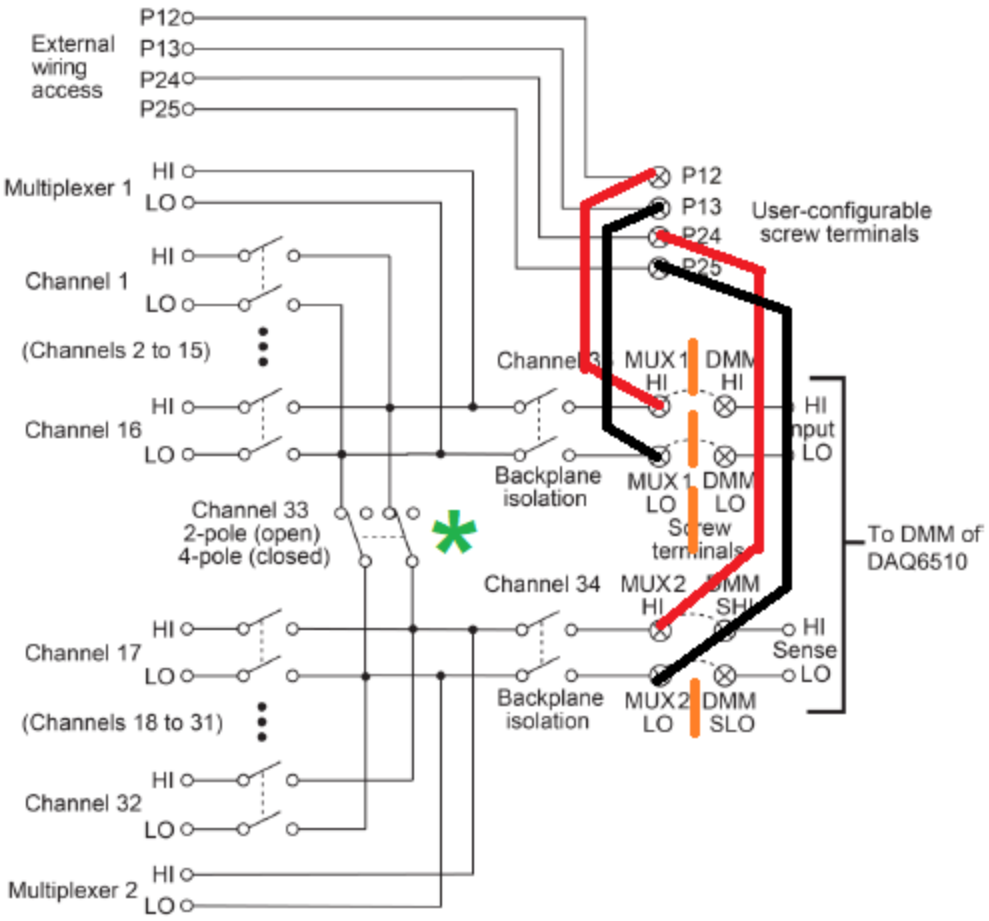
Figure 2: 7701 screw terminals



Notice how, in the simplified schematic below, this modification augments the connections scheme of the Model 7701 multiplexer.

- The DMM inputs are still disconnected
- The Multiplexer inputs are the primary switch bank common entry points
- Additional external wiring access is enabled via the user-configurable screw terminals

Figure 10: 7701 simplified schematic



Also note from the simplified schematic above that, while we no longer need to concern ourselves with DMM input connections, the MUX1 and MUX2 switch banks are not fully isolated from each other due to the normally closed (NC) state of the Channel 33 relay (near the green asterisk). The operator will need to close this channel in order to separate the MUX1 and MUX2 switch banks. Steps to close Channel 33 are included in the following sections.

Front Panel Operation

Enabling Full Channel Access

The DAQ6510 user interface provides a Channel user interface (UI) to allow for general open/close of all available channel relays. Additional control of the relays controlling the 2W-to-4W measure switching (channel 33) and the relays that connect the channels to the internal DMM Input and Sense terminals (channels 34 and 35) is available but first must be enabled.

To enable full relay channel access:

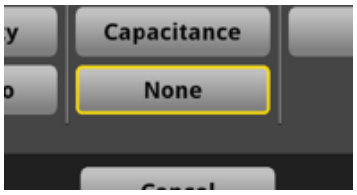
- Navigate to **MENU->System->Settings**
- Scroll to the bottom of the UI to locate the **Channel Access** control and touch the adjacent button.

- Select **Full** from the options provided.

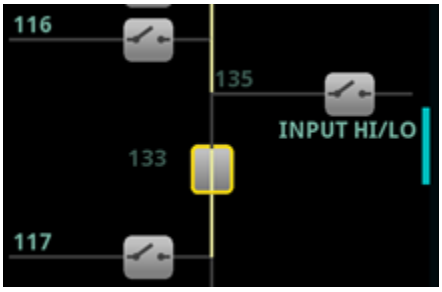
Controlling the Channel Paths

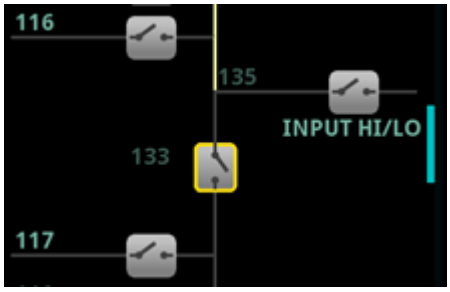
Opening and closing the desired channel paths can be accomplished through the instrument front panel. A brief example of how to do this is as follows:

1. Navigate to **MENU->Channel->Control** to reveal the interactive relay control tools.
2. Touch the (channel) **101** button.
3. When presented with the Measure Functions dialog, select the **None** option. The relay will issue the “click” noise and the UI will update to show the relay icon for the channel closed along with a highlighted closure path.



4. To open the relay that connect the MUX1 and MUX2 relay banks, scroll down in the UI until (channel) **133** is revealed. Note how it shows a closed state.
5. Touch the (channel) **133** icon to open. The relay will issue a “click” noise and the UI will update to show the relay icon for the channel open along with the highlighted path disrupted.





6. Note in the images above the presence of the (channel) **135** relay which, on the Model 7701 module, would be used to connect MUX1 to the DMM INPUT HI/LO terminals.

Remote Operation

To help illustrate the concept of using the DAQ6510 solely as a switch the following remote programming example is proposed:

- A Model 7701 is seated in slot 1 of the DAQ6510 mainframe.
- A power source is applied to MUX2 and routed to any of three devices connected to channels 117, 118, and 119 of the Model 7701
- A measurement device is applied to MUX1 and routed to any of six possible test points connected to channels 101, 102, 103, 104, 105, and 106 of the Model 7701.
- The jumpers between MUX1 and the DMM Input terminals and those between MUX2 and the DMM Sense terminals have been removed from the Model 7701. However, channels 134 and 135 are forced to their open state to help ensure isolation.
- The MUX1 and MUX2 multiplexer banks are separated by closing channel 133.
- For each MUX2 source connection to the device, each of the six MUX1 channels will be closed to allow for a measurement to occur then open to allow for subsequent channel measurements.

The following Python code snippet provides the user the command calls necessary to execute the channel actions that facilitate the proposed test. For the full set of control code along with supporting sockets-based communications functions, refer to **Appendix A**. Note that while we are using the TSP command set here, the DAQ6510 also offers a SCPI command set with equivalent functionality.

```
source_channel_list = (["117", "118", "119"])
measure_channel_list = (["101", "102", "103", "104", "105", "106"])

# open channels 134 and 135 to supplement isolation from the DAQ6510
# internal input and sense terminals
instrument_write(my_daq, "channel.multiple.open(\"134,135\")")

# close channel 133 to ensure the disconnect between the MUX1 and MUX2
# multiplexer banks
instrument_write(my_daq, "channel.multiple.close(\"133\")")

for s_chan in source_channel_list:
    # close the source channel
    instrument_write(my_daq, "channel.multiple.close(\"{0}\")".format(s_chan))
    for m_chan in measure_channel_list:
        # close the test point measurement channel
        instrument_write(my_daq, "channel.multiple.close(\"{0}\")".format(m_chan))

        # insert a delay representative of the measurement activity performed
        # by the external instrument
        time.sleep(0.5)

        # open the test point measurement channel
```

```
        instrument_write(my_daq, "channel.multiple.open(\"{0}\")".format(m_chan))

    # open the source channel
    instrument_write(my_daq, "channel.multiple.open(\"{0}\")".format(s_chan))

    # add another delay representative of the external source management
    #commands
    time.sleep(0.5)
```

Summary

The DAQ6510 is far more versatile than to act as a datalogger alone. While this paper has focused on a switching-only scenario focused primarily on use with the Model 7701, almost all the content is applicable to any of the seven other Keithley modules that offer the multiplexing topology. (See **Appendix B** for a list of multiplexer module options.) This includes front panel control of individual relays via the rich touchscreen UI as well as the programming commands.

Appendix A

Full programming example for customized switching control.

```
"""************************************************************
*** Copyright Tektronix, Inc.                                ***
*** See www.tek.com/sample-license for licensing terms.      ***
************************************************************"""
import socket
import struct
import math
import time

echo_cmd = 0

"""*****************************************************************************
   Function: instrument_connect(my_socket, ip_address string, my_port int, timeout
                                do_reset, do_id_query)

   Purpose: Open an instance of an instrument object for remote communication
       over LAN/Ethernet.

    Parameters:
    my_socket - Instance of a socket object.

      ip_address (string) - The TCP/IP address string associated with the
                            target instrument.
      my_port (int) - The instrument connection port.

      timeout (int) - The timeout limit for query/communication exchanges.

      do_reset (int) - Determines whether the instrument is to be reset
                       upon connection to the instrument. Setting to 1
```

```
                        will perform the reset; setting to zero avoids it.

      do_clear (int) - Determines whether the instrument is to be cleared


        do_id_query (int) - Determines when the instrument is to echo its
                            identification string after it is initialized.

   Returns:
      my_socket - Updated instance of a socket object that includes
                            attributes of a valid connection.

   Revisions:
      2019-07-30    JJB    Initial revision.
**************************************************************************"""


def instrument_connect(my_socket, my_address, my_port, timeout, do_reset, do_clear,
do_id_query):
    my_socket.connect((my_address, my_port))  # input to connect must be a tuple
    my_socket.settimeout(timeout)
    if do_reset == 1:
        instrument_write(my_socket, "*RST")
    if do_clear == 1:
        instrument_write(my_socket, "*CLS")
    if do_id_query == 1:
        tmp_id = instrument_query(my_socket, "*IDN?", 100)
        print(tmp_id)
    return my_socket


"""**************************************************************************
   Function: instrument_disconnect(my_socket)

   Purpose: Break the LAN/Ethernet connection between the controlling computer
          and the target instrument.

   Parameters:
      my_socket - The TCP instrument connection object used for sending
                      and receiving data.

   Returns:
      None

   Revisions:
      2019-07-30    JJB    Initial revision.
**************************************************************************"""


def instrument_disconnect(my_socket):
    my_socket.close()
    return


"""**************************************************************************
   Function: instrument_write(my_socket, my_command)

   Purpose: This function issues control commands to the target instrument.

   Parameters:
      my_socket - The TCP instrument connection object used for sending
             and receiving data.
      my_command (string) - The command issued to the instrument to make it
```

```
                perform some action or service.
   Returns:
      None

   Revisions:
      2019-07-30    JJB    Initial revision.
**************************************************************************"""


def instrument_write(my_socket, my_command):
    if echo_cmd == 1:
        print(my_command)
    cmd = "{0}\n".format(my_command)
    my_socket.send(cmd.encode())
    return


"""**************************************************************************
   Function: instrument_read(my_socket, receive_size)

   Purpose: This function asks the connected instrument to reply with some
               previously requested information, typically queued up from a call
               to instrument_write().

   Parameters:
      my_socket - The TCP instrument connection object used for sending
             and receiving data.
      receive_size (int) - Size of the data/string to be returned to
                                  the caller.

   Returns:
      reply_string (string) - The requested information returned from the
               target instrument.

   Revisions:
      2019-07-30    JJB    Initial revision.
**************************************************************************"""


def instrument_read(my_socket, receive_size):
    return my_socket.recv(receive_size).decode()


"""**************************************************************************
   Function: instrument_query(my_socket, my_command, receive_size)

   Purpose: This function issues control commands to the target instrument with
               the expectation that data will be returned. For this function
               instance, the returned data is (typically) in string format.

   Parameters:
      my_socket - The TCP instrument connection object used for sending
             and receiving data.
      my_command (string) - The command issued to the instrument to make it
                perform some action or service.
      receive_size (int) - The approximate number of bytes of data the caller
                                  expects to be returned in the response from the
                                  instrument.
   Returns:
      reply_string (string) - The requested information returned from the
               target instrument. Obtained by way of a caller
               to instrument_read().
```

```
   Revisions:
      2019-07-30    JJB    Initial revision.
********************************************************************************"""


def instrument_query(my_socket, my_command, receive_size):
    instrument_write(my_socket, my_command)
    return instrument_read(my_socket, receive_size)


"""********************************************************************************

   This example application demonstrates the use of the DAQ6510 in a switching
   only application. The channel control reflects the use of the Model 7701
   multiplexer card. The setup assumes the following:
       A. A power source is applied to MUX2 and routed to any of three devices
          connected to channels 117, 118, and 119.
       B. A measurement device is applied to MUX1 and routed to any of six possible
          test points connected to channels 101, 102, 103, 104, 105, and 106.
       C. The jumpers between MUX1 and the DMM Input terminals and those between
          MUX2 and the DMM Sense terminals have been removed from the Model 7701.
          However, relays 34 and 35 are forced to their open state to help ensure
          isolation.
       D. For each MUX2 source connection to the device, each of the six MUX1
          channels will be closed to allow for a measurement to occur then
          open to allow for subsequent channel measurements.

********************************************************************************"""
my_ip_address = "192.168.1.67"  # Define your instrument's IP address here.
my_port = 5025  # Define your instrument's port number here.

do_instr_reset = 1
do_instr_clear = 0
do_instr_id_query = 1

t1 = time.time()

# Open the socket connections...
my_daq = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Establish a TCP/IP
socket object
my_daq.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
instrument_connect(my_daq, my_ip_address, my_port, 20000, do_instr_reset,
do_instr_clear, do_instr_id_query)

source_channel_list = (["117", "118", "119"])
measure_channel_list = (["101", "102", "103", "104", "105", "106"])

# open channels 134 and 135 to supplement isolation from the DAQ6510
# internal input and sense terminals
instrument_write(my_daq, "channel.multiple.open(\"134,135\")")

# close channel 133 to ensure the disconnect between the MUX1 and MUX2
# multiplexer banks
instrument_write(my_daq, "channel.multiple.close(\"133\")")

for s_chan in source_channel_list:
    # close the source channel
    instrument_write(my_daq, "channel.multiple.close(\"{0}\")".format(s_chan))
    for m_chan in measure_channel_list:
        # close the test point measurement channel
        instrument_write(my_daq, "channel.multiple.close(\"{0}\")".format(m_chan))

        # insert a delay representative of the measurement activity performed
```

```
            # by the external instrument
            time.sleep(0.5)

            # open the test point measurement channel
            instrument_write(my_daq, "channel.multiple.open(\"{0}\")".format(m_chan))

        # open the source channel
        instrument_write(my_daq, "channel.multiple.open(\"{0}\")".format(s_chan))

        # add another delay representative of the external source management
        #commands
        time.sleep(0.5)

instrument_disconnect(my_daq)

t2 = time.time()  # Stop the timer...

# Notify the user of completion and the data streaming rate achieved.
print("done")
print("Total Time Elapsed: {0:.3f} s".format(t2 - t1))

input("Press Enter to continue...")
exit()
```

Appendix B

Other Keithley switch modules with multiplexing functionality:

- Model 7700 20-channel Differential Multiplexer Module (with automatic CJC)
- Model 7701 32-channel Differential Multiplexer Module
- Model 7702 40-channel Differential Multiplexer Module
- Model 7703 32-channel High-speed Differential Multiplexer Module
- Model 7706 All-in-one I/O Module (20-channel differential multiplexer with automatic CJC, 16 digital outputs, 2 analog outputs, and counter/totalizer)
- Model 7707 32-bit Digital I/O Module (with 10-channel differential multiplexer)
- Model 7708 40-channel Differential Multiplexer Module (with automatic CJC)
- Model 7710 20-channel Solid-state Differential Multiplexer Module (with automatic CJC)