KEITHLEY A GREATER MEASURE OF CONFIDENCE

Creating External Instrument Drivers for the Model 4200-SCS

Introduction

As the measurements performed on semiconductor devices grow increasingly complex, so does the demand for measurement automation. A typical test setup often involves several instruments performing sourcing, measurement, or auxiliary functions, all connected to a common communications bus (typically GPIB) and controlled by a PC station. The Model 4200-SCS parameter analyzer eliminated the need for a dedicated PC. The Keithley Interactive Test Environment (KITE) allows the Model 4200-SCS to perform as both a parameter analyzer and an external instrument controller, making it the "command-and-control center" of the entire instrument rack. KITE software already supports several popular instruments, such as pulse generators, switch matrices, and C-V analyzers, through software control modules known as *drivers*.

Occasionally, a user may need to control an instrument that's not supported by a standard Keithley driver library. This technical note describes how to create *custom instrument drivers*. It assumes the reader is already familiar with the basic operation and software environment of the Model 4200-SCS, as well as with programming in C language.

Overview

In general, the Model 4200-SCS can control any external instrument connected to either the IEEE-488 (GPIB) bus or the RS-232 (serial) communication port, thanks to its flexible PC-based architecture. The GPIB protocol is currently the most widely used one. Figure 1 illustrates a multi-instrument system configuration.

In the Model 4200-SCS, external equipment is controlled via the User Test Modules (UTMs), which are essentially C functions created and maintained with the Keithley User Library Tool (KULT). Figure 2 illustrates the relationship between user modules, user libraries, UTMs, KITE, and KULT.

From an operator's perspective, controlling an external instrument via a UTM is very similar to programming that instrument from its own front panel. Once all the instrument settings are entered and saved, running an instrument is as easy as clicking the Run button in KITE. The added advantage of external control, however, is that now this UTM can be incorporated into a larger KITE project and become a part of a 4200-based automated test setup. Data collection, processing, graphing, and storage are also simplified.

Creating External Instrument Drivers for the 4200-SCS

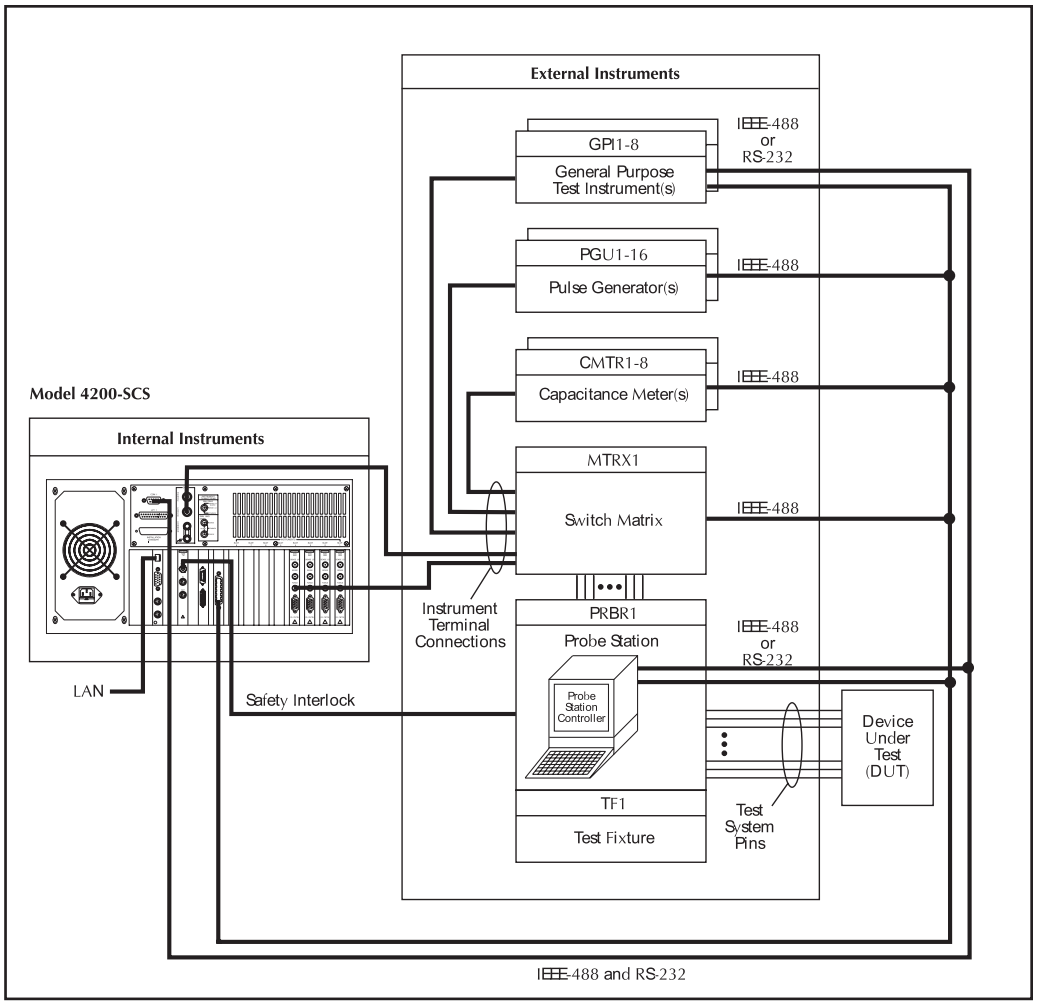

Figure 1.

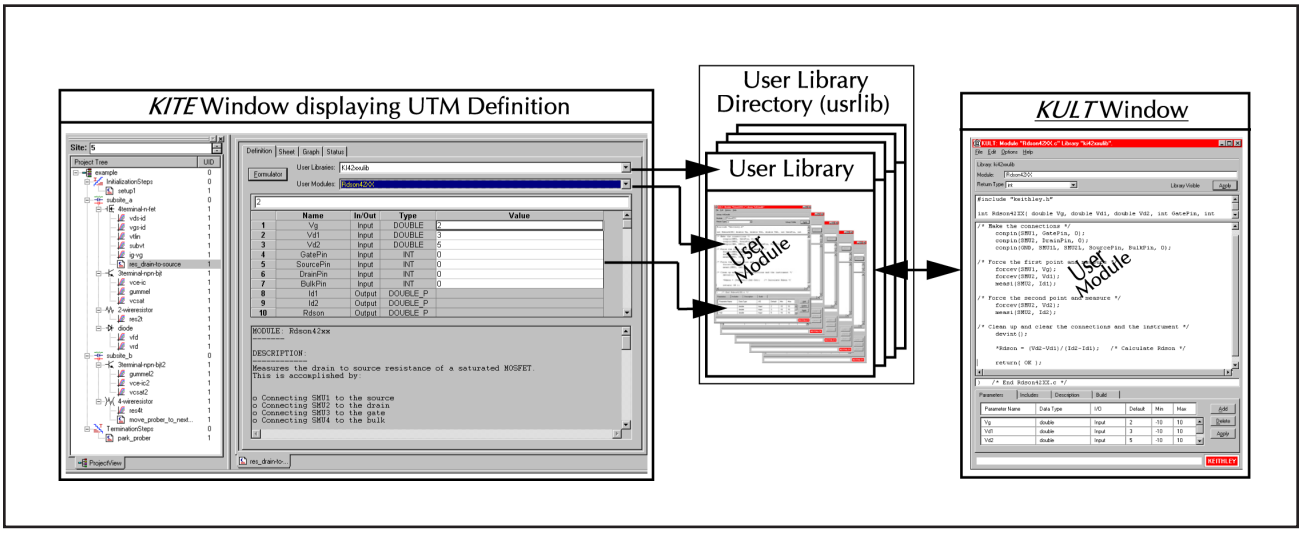

Figure 2.

Driver Structure

An instrument driver can be as simple or as complex as the function the instrument is asked to perform. Typically, external instruments fall into two general categories: *auxiliary devices* (e.g., wafer probers, thermal chuck controllers, pulse generators, switch mainframes) and *measurement instruments* (e.g., C-V meters, LCR meters, oscilloscopes).

Auxiliary device drivers are typically structured as illustrated in Figure 3.

Note that the auxiliary devices require mostly one-way communication from the controller, which makes the drivers very simple. Measurement instruments perform more complex functions; therefore, their drivers are usually more complex. A typical measurement instrument driver is structured as illustrated in see Figure 4.

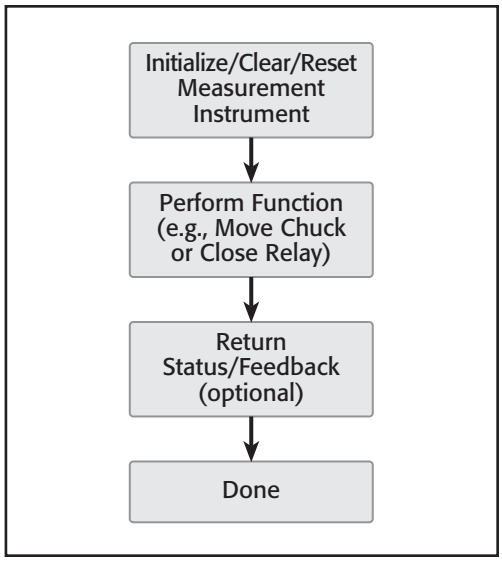

Figure 3.

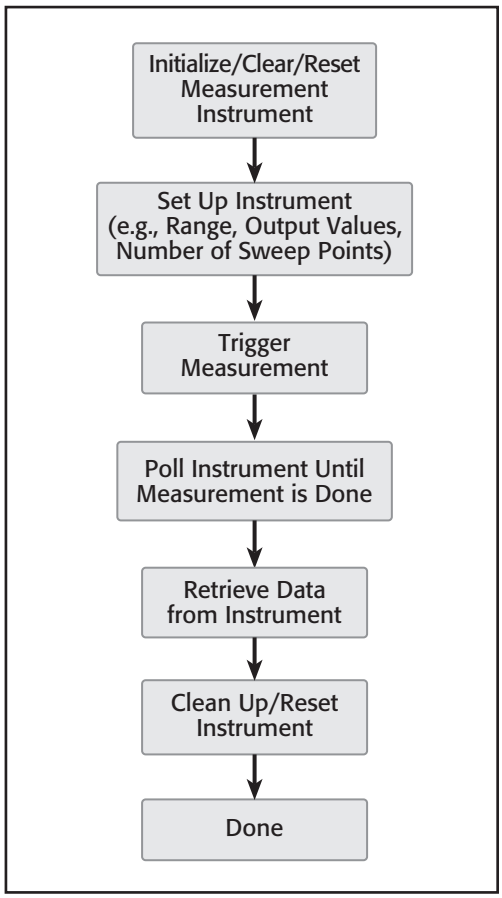

Figure 4.

General Considerations

Let's look at an example from the standard Keithley libraries (the existing libraries that come standard on every Model 4200-SCS).

The standard library for semi-automatic probers on the Model 4200-SCS is called *PrbGen* (for Prober Generic). This library contains four modules: *PrInit* (Prober Initialize), *PrChuck* (Prober Chuck), *PrMovNxt* (Prober Move Next), and *PrSSMovNxt* (Prober SubSite Move Next). Without getting into the details of implementation of these modules, it's easy to see this library addresses three primary functions of a semi-automatic prober:

- Raising/lowering the wafer chuck (module *PrChuck*)
- Moving the chuck to the next site (die) on the wafer (module *PrMovNxt*)
- Moving the chuck to the next subsite (sub-die) on the wafer (module *PrSSMovNxt*)

It also addresses several auxiliary (but important) functions:

- Initializing the prober (module *PrInit*)
- Returning current X,Y die position after every move (module *PrMovNxt*)
- Returning an execution OK status or an error code (all modules)

This library illustrates several important points in the general approach to driver structure and implementation:

- ***Select only those functions that are necessary and important.*** Keep in mind that programmable instruments are usually capable of a much wider array of functions than is required for a given project/task, and trying to cover 100% of the instrument's capability can be a huge undertaking. In the semi-automatic prober example, a typical prober can easily have more than a hundred commands in its programming manual, while less than ten were sufficient to implement the library.

Creating External Instrument Drivers for the 4200-SCS

- ***Keep the driver modular as much as possible***. If there is a particular instrument function that may need to be called separately from accompanying functions, create a module around this function alone, rather than including it in a larger multi-function module. For example, the function of raising/lowering the chuck (*PrChuck*) could have been implemented as part of the function of moving the chuck to the next die (*PrMovNxt*) to form a "*Chuck Down->Move->Chuck Up*" sequence, but that would make it impossible to address the chuck without causing a move to the next wafer site.

Driver Implementation

Creating a new instrument driver usually involves the following stages:

1. **Planning**. The main goal of this stage is to develop a clear understanding of the functionality expected from the driver. By now, the user must be familiar with the operation of the instrument itself.
2. **Layout**. The goal of this stage is to break down the list of tasks/functions defined in Step 1 into functional modules. Also, identify all the corresponding remote commands, using the instrument's manual.
3. **Coding**. Here is where the layout created in Step 2 is implemented in C code using the KULT interface. Includes initial code debugging to get the library to compile and build.
4. **Testing**. At this stage, with the instrument connected to the 4200-SCS, the driver is tested to see if it works as expected.
5. **Debugging and adding on**. If the testing in Step 4 revealed some coding flaws, they need to be identified and repaired, and the driver re-tested. Once the driver starts working, many users decide to make additions and improvements to the existing code.
6. **Creating documentation**. This is an important but often forgotten step—it allows the author of the driver to communicate how the driver works to other users, and makes future debugging and code maintenance much easier. May take the form of comments within the source code, user/operator instructions entered in the Description area of the KULT window, a stand-alone Readme file, or all of the above (recommended).

Implementation Example

We will now follow the step-by-step process of creating a simple auxiliary device driver, using Keithley's Model 7002 switch matrix as an example of the external instrument.

The primary function of a switch matrix is to form interconnections according to the desired pattern, so we will limit the driver to two basic user modules: one that performs initialization and one that closes a single row-column crosspoint.

Open the KULT interface by double-clicking the KULT icon on the Model 4200-SCS desktop. A blank KULT window appears named KULT: Module "NoName" Library "NoName". Select *File -> New Library*, and enter name **Keithley _ 7002 _ switch**. Then select *File -> New Module*, and enter the name **Initialize _ switch**.

From a C language programming standpoint, so far, we have written the following:

```
#include "keithley.h"

void Initialize _ switch (    )
{

}     /* End Initialize _ switch.c*/
```

The first line, **#include "keithley.h**", is a so-called preprocessor directive, which instructs the C compiler to include functions not explicitly defined in our module. This is inserted in the code automatically by KULT, and keeping this line is recommended. The next line, **void Initialize _ switch (   )**, declares the name of the function as well its return type. The curly braces indicate the start and end of the empty body of the function. The line **/*End Initialize _ switch.c*/** is a comment, because the C compiler interprets anything between **/*** and ***/** as a comment. Please note that all of these elements of the code are created automatically by the KULT environment, and need not be manually entered.

These steps have created a framework for the new driver, and we are now ready to start programming. In the module programming area, enter:

```
char command _ string[20];

sprintf(command _ string, "*RST");
kibsnd(GPIB _ address, -1, GPIBTIMO, strlen(command _ string), command _ string);
```

The first line, **char command _ string[20];**, declares an internal variable of type string, named **command _ string**, with a maximum length of 20 characters. The C compiler requires all variables to be declared at the beginning, before calling any functions or performing any other operations. It is okay give this string variable any other name, as long as the name is used consistently throughout the rest of the module.

The second line, **sprintf(command _ string, "*RST");**, assigns the string "*RST" to the variable **command _ string**. The **sprintf(   )** is a standard C function. The ***RST** is a GPIB command, which, according to the Model 7002 Switch System Instruction Manual, "returns the Model 7002 to the *RST default conditions." It's always a good idea to put an instrument into a known state before programming it. In this case, the *RST clears all the errors and opens all the crosspoints on the 7002 switch, and this is exactly what we want to achieve with this initialization module.

The third line, **kibsnd(GPIB _ address, -1, GPIBTIMO, strlen(command _ string), command _ string);**, calls function **kibsnd(   )**, which instructs the Model 4200-SCS to send a string contained in the variable **command _ string** to the GPIB instrument at address **GPIB _ address**. The function **kibsnd(   )** is part of the Keithley

Creating External Instrument Drivers for the 4200-SCS

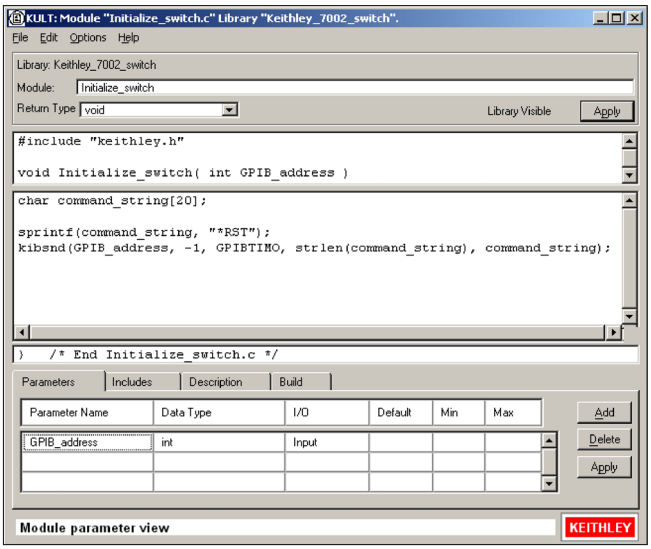

Figure 5.

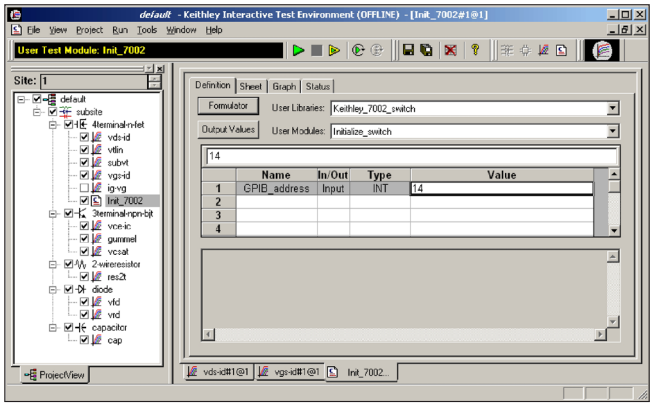

Figure 6.

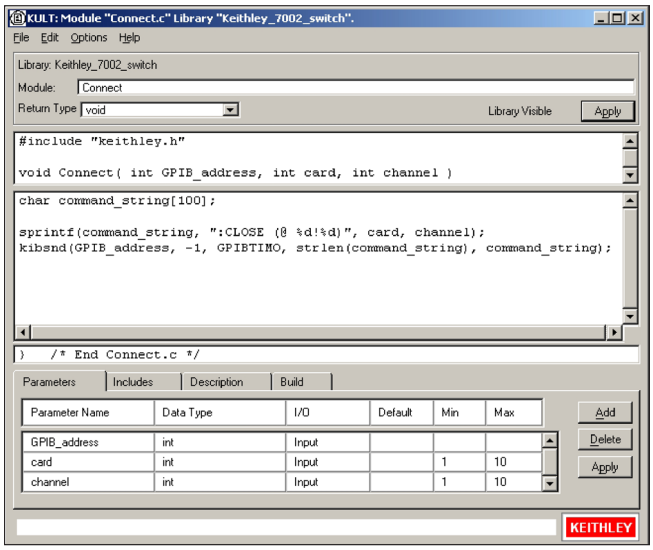

Figure 7.

Linear Parametric Test Library (LPTlib), which is documented in detail in the Model 4200-SCS Reference Manual.

Note that we have used a variable called **`GPIB _ address`** as the first argument of the **`kibsnd( )`** function. We will turn this variable into a *user parameter*, which can be edited from the KITE interface without making any changes to the source code. Select the Parameters tab at the bottom of the KULT window, click the Add button, and add this user parameter, as shown in Figure 5. This completes the code entry. The module needs to be saved (File -> Save Module), compiled (Options -> Compile), and built into a library (Options -> Build Library) before it can be executed.

It may be a good idea at this point to perform an actual test of the operation of this simple module, before more coding is done. After connecting the 7002 switch to the Model 4200-SCS with a GPIB cable, create a new UTM in KITE and point it to the new module, as shown in Figure 6.

Pressing the Run button in KITE should reset the 7002 switch to a default state. If this doesn't happen, take steps to determine and correct the cause of the failure (e.g., poor cable connection, wrong GPIB address entered, etc.)

Once the communication had been established and the instrument responds the way it should, the driver can be completed by adding the second module, which performs the actual function of closing a crosspoint in the switch matrix. Following the same steps as previously, we will arrive at the following like that shown in Figure 7. The driver is now complete.

Driver implementation – Beyond the Basics

Please keep in mind that this example was chosen to illustrate the process of creating the simplest driver. Most instruments typically require setting up a dozen or more commands properly. Drivers may also include error checking, operator prompts, conditional statements, and many other features that are possible in the programming environment. Some of the additional techniques are summarized in the following subsections, using fragments of simplified driver code, borrowed largely from the standard libraries. The standard 4200-SCS libraries are a great source of programming examples and techniques. Readers are highly encouraged to study those drivers and borrow from them to speed up the learning curve and achieve results in the most efficient manner.

Initial "handshaking"

It's sometimes a good idea to have the driver detect the instrument and determine its model before proceeding with further programming. If an unsupported model is detected, the driver should output an error message and quit. The following fragment of code illustrates this technique in the case of the Keithley Model 2400 driver:

```
sprintf(CommandString, "*IDN?\n");          // Send a request for instrument model ID
kibsnd(GPIBAddress24xx, -1, GPIBTIMO, strlen(CommandString), CommandString );
kibrcv(GPIBAddress24xx, IGNORE _ PARAM, LF, GPIBTIMO, max _ size, &rcv _ size, RawReading);
                                            // Receive instrument ID string

    if(strstr(RawReading,"2400") = = NULL) // if it is not a 2400
    {
        return ("Instrument not recognized");
    }
```

Serial polling

An instrument that had been programmed to perform a certain function (e.g., make a measurement) may take an indeterminate amount of time to complete it and return results. To indicate that the task has been completed, instruments set an SRQ status bit. Reading the status of the SRQ bit is known as serial polling. The following code fragment illustrates how this is done in the Model 590 driver:

```
// Set up to enable SRQ on sweep complete
sprintf(CommandString, "M4X" );
kibsnd(GPIBAddress, IGNORE _ PARAM, GPIBTIMO, strlen(CommandString), CommandString);
…….
// Serial poll the instrument to clear
kibspl(GPIBAddress, IGNORE _ PARAM, GPIBTIMO, &spollbyte);

// Now serial poll until the measurement is complete
kibsplw(GPIBAddress, IGNORE _ PARAM, GPIBTIMO, &spollbyte);
```

Returning measurement data

The ultimate goal of a measurement instrument driver is to obtain and bring the data into KITE for plotting and analysis. Depending on the instrument and the type of measurement, the format of the returned data will vary. A single-point measurement will return a scalar, while a sweep will return an array (or several arrays). These return parameters must be defined in a UTM as output-type arguments of appropriate type. Typically, after the measurement is complete, as indicated by the set SRQ bit, the instrument needs to be addressed to return data. The following code example illustrates how array data is obtained from a Model 590:

Creating External Instrument Drivers for the 4200-SCS

```
// Set up a loop:
for (index=0; index < num_readings; index++)
    {
                                    // Read a set of data:
kibrcv(GPIBAddress, IGNORE_PARAM, LF, GPIBTIMO, max_size, &rcv_size, RawReading);
                                    //Convert string into three numbers:
         sscanf(RawReading, "%11g,%11g,%11g", &temp1, &temp2, &temp3);
                                    // Assign raw reading data to output arguments:
         C[index] = temp1;
         G_or_R[index] = temp2;
         V[index] = temp3;
     }
```

Outputting messages

There are several ways to make a UTM output messages, just as there may be several reasons for a programmer to want to output a message. During code debugging, it might make sense to output text to a Keithley Message Console, which is enabled by typing **msgcon** at the MS-DOS command prompt. Once the code is working, it might be better to have the module return its status as a string—for example, "OK" or "ERROR: GPIB TIME-OUT." Finally, a pop-up dialog may be called from within a module, using existing dialog types from the *winulib* library. These three methods are illustrated here:

```
.....
// Output a message to the Keithley Message Console:
printf("measurement complete");

.....
// Output a message for an operator through an OK-type dialog pop-up window:
OkDialog(1, "hello world", "", "", "");

.....
//Return function execution status:
return ("OK");
```

Conclusion

The KULT interface expands the built-in capabilities of the Model 4200-SCS well beyond those of a dedicated parameter analyzer. Custom instrument drivers address the need for measurement system integration by providing users with a wide array of programming options.

KEITHLEY

Keithley Instruments, Inc.
28775 Aurora Road • Cleveland, Ohio 44139 • 440-248-0400 • Fax: 440-248-6168
1-888-KEITHLEY (534-8453) • www.keithley.com

No. 2661
1005