HEWLETT PACKARD

Object-Oriented Techniques for C Programmers
A Series of Test & Measurement Examples

Jeff Burch

Measurement Systems Department
Instrument and Photonics Lab
HPL-95-07
January, 1995

Abstract

This paper introduces object-oriented (OO) programming techniques to the C programmer. Unlike most books on this subject that immediately launch into OO jargon, this paper leads the reader through a series of increasingly complex procedural programming examples. In each step, realistic problems are presented and improvements to an example C program are introduced. In the final step, the program is re-written in C++.

All examples in this paper come from an hypothetical measurement system application. Some experience in building systems to control instruments and familiarity with making measurements is assumed.

The document concludes with a concise summary of techniques that C programmers can use today without switching to an object-oriented language. However, the document also stresses that using the C++ compiler to manage the necessary initialization details is warranted.


HEWLETT
PACKARD

Table of Contents

1 Introduction

This paper introduces object-oriented (OO) programming techniques to the C programmer. Unlike most books on this subject that immediately launch into OO jargon, this paper leads the reader through a series of increasingly complex procedural programming examples. In each step, realistic problems are presented and improvements to an example C program are introduced. In the final step, the program is rewritten in C++.

All examples in this paper come from a hypothetical measurement system application. Some experience in building systems to control instruments and familiarity with making measurements is assumed. The following techniques are illustrated: (1) the use of standard interfaces for the instrument driver software layer, (2) support for instrument substitution, (3) the management of state variables to allow more than one instrument of the same type, (4) using jump tables to simplify invoking the instrument driver routines from the measurement layer, and (5) management of initialization details.

2 Acknowledgments and Background

None of the programming techniques used are original to this paper . Experienced C programmers will feel at home with mechanisms like jump tables, standard interfaces, and state structs. Therefore, I thank all of you who invented these techniques and have been using them for years.

Credit is also due to my former managers at the HP Opto-Electronics Division (OED) in San Jose, Kim Harris and Jan Unter. Through their guidance, vision, and ability to secure and obtain necessary human resources, they launched an innovative object-oriented system architecture project for use in manufacturing test applications of which I had the good fortune to be a part.

Much credit is also due to my system architect coworkers, Regan Fuller and Lucy Lin. Through many meetings, discussions, arguments, and false starts, we succeeded to teach each other a great many object-oriented programming techniques. More importantly, we were the collective architects for the above mentioned object-oriented application test framework.

Thanks are also due to my fellow programmers in OED who suffered through a classroom version of this document during the spring of 1991. Basically, this class and document resulted during a period of financial uncertainty for the division with the clear threat of immediate downsizing (i.e., I was next on the list to be excessed!), when I convinced management to let me teach an internal object-oriented programming class to coworkers.

Because there were plenty of experienced C programmers who were very skeptical of most object-oriented jargon, I was forced to come up with a new approach to the material. I would have probably lost my audience if I started the class by defining and illustrating new object-oriented concepts.

Rather than launching into topics like inheritance, encapsulation, and the proper use of new keywords like "virtual," I walked the group through a series of procedural-based measurement system examples like those discussed in this paper.

The outcome of this class was very positive. Most of the students completed the class with a clearer understanding of object-oriented concepts, why they were useful, and how they could be applied to their work. Also, most left with an appreciation that changing to an object-oriented language (like C++ that will hide and simplify the required initialization details) was well worth the effort.

For those readers who have a similar procedural-based persuasion, hopefully you will find this "series of examples" format helpful. Perhaps, you may even be persuaded to try your next project in an object-oriented language!

3 Organization

This document is organized as follows:

An overview of the measurement set-up and the device to be tested is given in Section 4.

In Section 5, a brief introduction to VXI instrumentation is given. A quick summary of register-based instrument control is presented.

In Section 6, the software architecture to support a single instrument is presented. Enough detail on test, measurement, and instrument driver software layers is given to give the reader the feel of such a system.

Our programming gets slightly more complicated in Section 7, where we substitute the old instrument for a new one. Just what has to change to accommodate the new instrument is highlighted.

The programming complexity increases in Section 8, when we need to accommodate both the new and old instruments. We introduce techniques to manage name collisions between similar procedures.

Our programming gets very tedious in Section 9, when we need to accommodate many similar instruments. The programming task takes on increasing amounts of drudgery just to manage the complexity. In other words, less effort is spent on creative programming tasks and more is consumed in just managing instrument differences.

In Section 10, our programming life becomes miserable when we need to manage more than one identical instrument at the same time. Techniques are introduced to provide unique copies of state variables for each instrument.

Finally, object-oriented techniques are introduced in Section 11 to handle problems in the preceding sections. The code is modified but is still written entirely in standard ANSI C. The portion of the code that performs "real work" is highlighted and the amount of "housekeeping" code is indicated.

In Section 12, the code is rewritten in C++. Hopefully, it will be clear to the reader that most of the housekeeping code is handled by the C++ compiler.

The document closes with sections on further directions and conclusions.

4 The Device Under Test

In all examples, the device under test (DUT) is a simple light-emitting diode (LED). Our measurement set-up is illustrated in Figure 1.

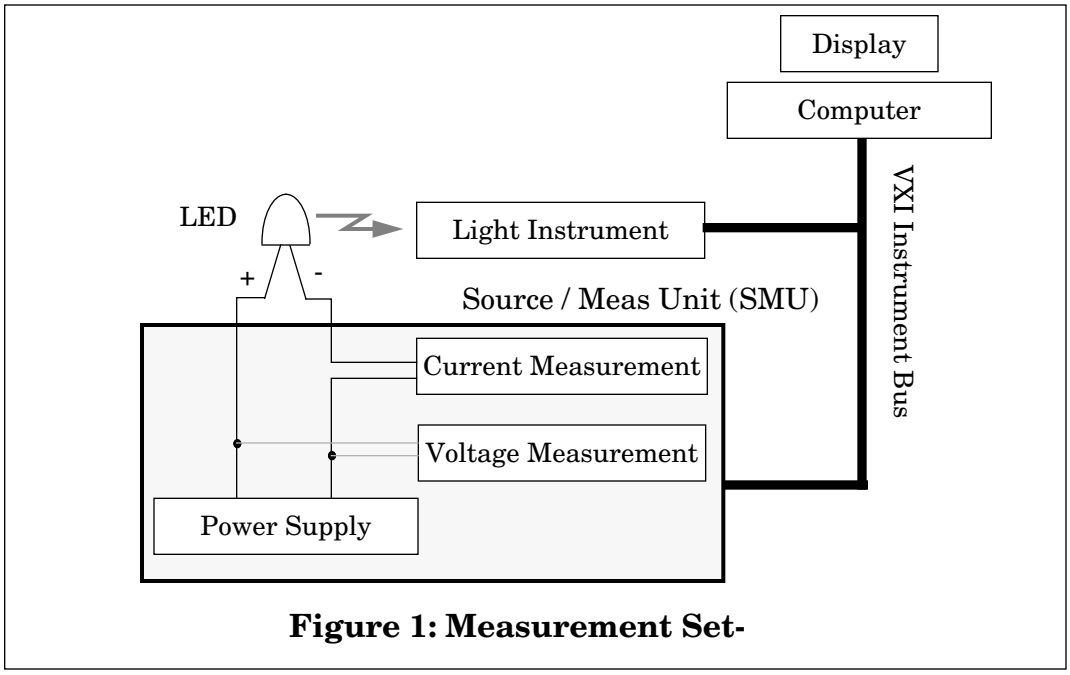

Figure 1: Measurement Set-

We have two instruments connected to the computer over an instrument bus. In this case, we assume VXI-based instrumentation. Under computer control, the source/ measurement unit (SMU, e.g., HPxxxx) outputs power to light the LED. For example, the computer may instruct the SMU to output 20 mA with a maximum voltage limit of 5 volts. After an appropriate warm-up delay, voltage, color, and light power measurements are made.

Internally, the SMU engages the power supply subsection and makes repeated current and voltage measurements to stay within the desired limits. Typical I/V characteristics for LEDs are illustrated in Figure 2.

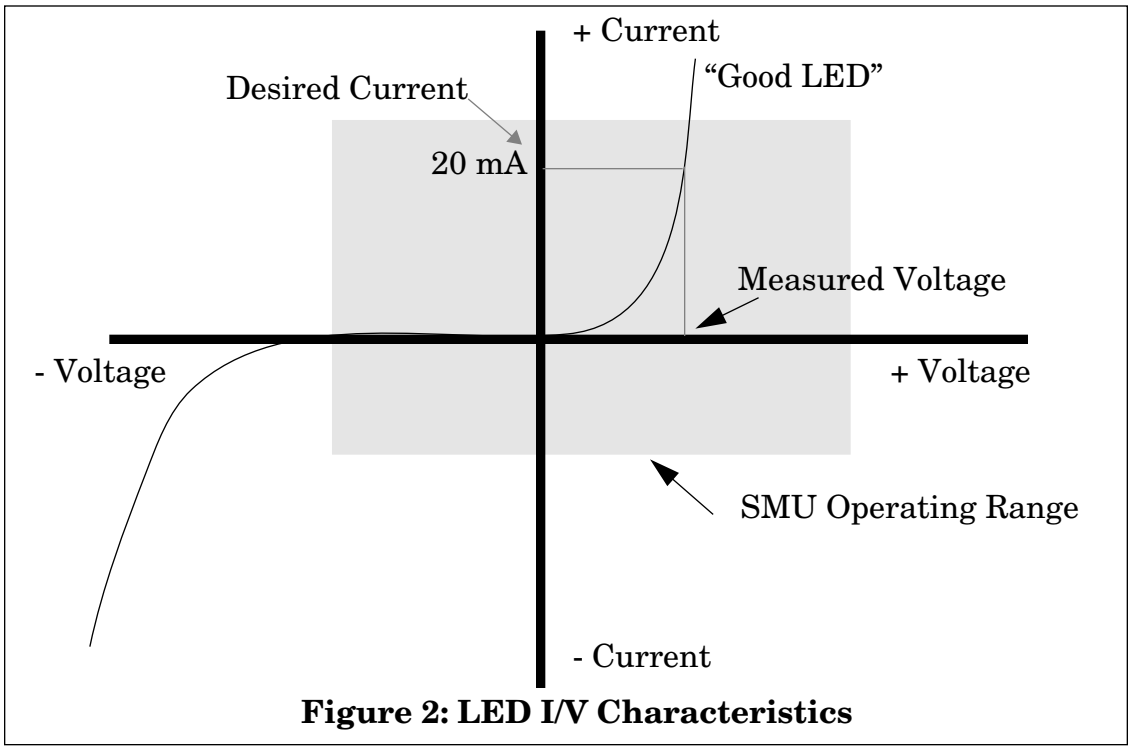

Figure 2: LED I/V Characteristics

5 Instrument Control over VXI

VXI is an instrument backplane based on VME. In this case, we assume simple register-based instruments and that access will be via simple register reads and writes.[1] Also, we simplify the discussion by concentrating only on the SMU. In reality, software would need to be developed to control and manage all instruments.

To control such a register-based instrument, the following information and I/O support library are needed:

- Details on the instrument's register map must be obtained. This includes (1) what each register does, (2) offsets for the registers from the instrument's base address, and (3) the proper sequence of register operations to make the instrument perform a function.
- The base address for the instrument must be obtained. In VXI, this is determined by which slot the instrument is plugged into.
- A mechanism is needed to read and write the appropriate register across the VXI backplane.

1. In practice, such a system could be built using SICL (Standard Instrument Control Library) and cSCPI (compiled Standard Commands for Programmable Instruments). Both products are available from HP's VXD division.

In this case, we assume the VXI I/O functions listed in Table 1.

Table 1: VXI I/O Functions

Function Prototype	Description
unsigned long VXI_get_base_address(int slot)	Get base add. for this slot
unsigned char VXI_read_byte(unsigned long address)	Read a 8-bit register
unsigned short VXI_read_short(unsigned long address)	Read a 16-bit register
unsigned long VXI_read_long(unsigned long address)	Read a 32-bit register
int VXI_write_byte(unsigned long address, unsigned char value)	Write a 8-bit register
int VXI_write_short(unsigned long address, unsigned short value)	Write a 16-bit register
int VXI_write_long(unsigned long address, unsigned long value)	Write a 32-bit register

6 Supporting a Single Instrument

In this section, a simple test system will be constructed that uses a single SMU instrument (called HPsmu_A). The task is to design appropriate software procedures for these functions:

Instrument Driver	Functions that manage instrument I/O details, e.g., source_on(), source_off(), source_current(), ...
Measurement Layer	Functions that manipulate the instruments (via instrument drivers) to perform useful measurements on the DUT, e.g., meas_led_color(), meas_shorts(), meas_voltage(), ...
Test Layer	Functions that control a sequence of measurements on the DUT and determine how the device is functioning. Also, the DUT may be compared with performance specifications, e.g., test_led(), bin_led(),...

To simplify the discussion, only one test and measurement function and a few driver-level functions will be presented. In the following subsections, example functions are presented to give the reader the flavor of how the software is partitioned among the various layers.

6.1 The test_led() function

The test_led() function is responsible for making light and voltage measurements at five different current settings. In essence, it cycles through the desired current values and invokes appropriate measurement-level functions. Inputs to the

function include the desired number of currents to test, their values, and specification information on the LED bin categories. Outputs from the function include voltage, color, and light power measurement results and LED bin classifications. (Note that error-checking has been omitted for clarity.)

```
/* test_led.c */
#include “meas_voltage.h”
#include “test_led.h”

int test_led(
      int num,                /* input, number of currents */
      double currents[],      /* input, array of current values */
      int led_categories[],   /* input, array of led bin classifications */
      double voltage[],       /* output, measured voltages */
      double color[],         /* output, measured colors */
      double light_power[],   /* output, measured light power */
      int led_bins[])         /* output, calculated LED bins */
{
      int i, status = TRUE;

      /* loop through the desired currents */
      for (i=0; i<num; i++)
            {
            /* make fundamental measurements */
            voltage[i] = meas_voltage( currents[i]);  /* source on */
            color[i] = meas_color( );
            light_power[i] = meas_light_power( );

            /* calculate LED bin from the fundamental measurements */
            led_bins[i] = led_bins( led_categories[i], color[i], ...);
            }

      meas_voltage( 0);       /* leave the source off */
      return (status);
}
```

6.2 The meas_voltage() function

The meas_voltage() function is a measurement routine that is responsible for configuring the SMU for the desired current and reporting back the current voltage value. (Note that the code is oversimplified and would need obvious improvements to prevent endless loops. Also, error checking has been omitted for clarity!)

```
/* meas_voltage.c */
#include “inst_dr.h”
#include “meas_voltage.h”

double meas_voltage( double current)
{
      double result;
      unsigned char range;
      unsigned short c_value, v_value;
```

```
        /* calculate the desired instrument current and range as shorts */
        calc_current_range( current, &range, &c_value);

        /* enable the driver */
        source_on( range, value);

        /* wait for the driver to stablize */
        while ( source_not_stable() );      /* Loop until stable */

        /* read the voltage and convert to double */
        v_value = source_voltage();
        result = calc_voltage_range( range, &v_value);

        /* disable the driver */
        source_off( );

        return (result);        /* return the voltage */
}
```

6.3 The source_on() function

The source_on() function is an instrument driver routine that is responsible for enabling the source hardware at the desired current level. For simplicity, only a few hardware registers are manipulated, register definitions are hard-coded, and error checking has been omitted. Also note the use of static local variables to manage instrument state.

Instrument driver “#include” file:

```
/* inst_dr.h */
/* defines for the h/w register offsets for the HPsmu_A SMU */

#define SMU_RANGE       0           /* Offset for 8-bit range register */
#define SMU_CONTROL     1           /* Offset for 8-bit control register */
#define SMU_CURRENT     2           /* Offset for 16-bit current reg */
#define SMU_VOLTAGE     4           /* Offset for 16-bit voltage reg */
...

/* driver state variables */
static unsigned long base_address;        /* Must initialize!!! */
static unsigned char control_mask;
static unsigned char range_value;         /* used for optimizations */
static unsigned short current_value;      /* used for optimizations */
...

/* function prototypes */
int source_init( int slot);
int source_on( unsigned char range, unsigned short current);
int source_not_stable();      /* Report stability */
double source_voltage();      /* Report source voltage */
...
```

Instrument driver “source” file:

```
/* inst_dr.c */
#include “inst_dr.h”

int source_init( int slot)
{
      /* Must initialize the base address to the correct VXI slot */
      base_address = VXI_get_base_address( slot);
      return (TRUE);
}

int source_on( unsigned char range, unsigned short current)
{
      /* optimization if we are at the desired conditions */
      if (range == range_value && current == current_value)
            return (TRUE);

      /* disable the source before we change the values */
      control_mask &= 0xf7;   /* clear the enable bit */
      VXI_write_byte( base_address+SMU_CONTROL, control_mask);

      /* set the desired range */
      if (range != range_value)
            {
            VXI_write_byte( base_address+SMU_RANGE, range);
            range_value = range;    /* save for next time */
            }

      /* set the desired current */
      if (current != current_value)
            {
            VXI_write_short( base_address+SMU_CURRENT, current);
            current_value = current;/* save for next time */
            }

      /* now enable source output */
      control_mask |= 0x08;   /* set the enable bit */
      VXI_write_byte( base_address+SMU_CONTROL, control_mask);

      return (TRUE);
}
```

6.4 Observations

Although the code is somewhat sketchy and only some routines are presented, it is fairly easy to design and build such a system using common C procedural programming techniques. A natural partitioning is presented among test, measurement, and instrument driver layers. Multiple functions are envisaged at each layer to perform the complete application, with each layer invoking the next-lower-layer routines. Although legal, skipping a layer (e.g., having the test_led() routine call the source_off() routine directly, to disable output) is not encouraged.

The management of state in the application is simple but not very extensible. The caller of the test_led() function was required to pass in parameters (e.g, led_categories()) that really are private data needed by the test. Although not shown, it is assumed that the caller is a specific test executive that can make this specific call. Note that other tests probably have a different call signature.

In addition, we assume that the test system integrator or operator has some mechanism to modify the desired test conditions. For example, the number of currents and their desired values can be modified.

In the instrument driver, private instrument state can be maintained in the static variables. This could be used to optimize the driver (e.g, to see if we are in the desired range before setting it) and minimize I/O to the instrument.

7 Simple Instrument Substitution

This section builds on the above example. Assume that we are cloning the test system and that the hp_smu_a SMU is no longer available. However, we can purchase a new and improved hp_smu_b SMU from the same HP division. We do not need to support the old SMU in the cloned system.

Because all instrument register access was restricted to the instrument driver layer, we propose to rewrite just those functions. We will not need to modify the measurement or test routines.

7.1 Modifications

We expect the new SMU to have a different hardware register map, additional features (e.g., higher output voltage or current ranges, or faster rise times). The inst_dr.h and source_on() routine are modified as follows:

Instrument driver “#include” file:

```
/* inst_dr.h */
/* defines for the h/w register offsets for the HPsmu_B SMU */

#define SMU_CONTROL     0           /* Offset for 8-bit control register */
#define SMU_FREQ        1           /* Offset for 8-bit freq control reg */
#define SMU_CURRENT     2           /* Offset for 32-bit current reg */
#define SMU_VOLTAGE     6           /* Offset for 32-bit voltage reg */
...

/* driver state variables */
static unsigned long base_address;              /* Must initialize!!! */
static unsigned char control_mask;
static unsigned char desired_operating_freq;    /* new state var */
static unsigned long current_value;             /* used for optimizations */
...

/* function prototypes */
int source_init( int slot);
int source_on( unsigned char range, unsigned short current);
```

```
int source_not_stable();      /* Report stability */
double source_voltage();      /* Report source voltage */
...
```

Instrument driver “source” file:

```
/* inst_dr.c */
#include “inst_dr.h”

int source_on( unsigned char range, unsigned short current)
{
      /* new SMU doesn’t have a range but takes 32 bit value */
      unsigned long c_value;

      /* backwards compatibility hack, get 32-bit current */
      c_value = unconvert_range( range, current);

      /* optimization if we are at the desired conditions */
      if (c_value == current_value) return (TRUE);

      /* disable the source before we change the values */
      control_mask &= 0xfe;   /* clear a different enable bit */
      VXI_write_byte( base_address+SMU_CONTROL, control_mask);

      /* set the desired current as a 32-bit value */
      VXI_write_long( base_address+SMU_CURRENT, c_value);
      current_value = c_value;/* save for next time */

      /* now enable source output */
      control_mask |= 0x01;   /* set the enable bit */
      VXI_write_byte( base_address+SMU_CONTROL, control_mask);

      return (TRUE);
}
```

7.2 Observations

Because the old instrument driver interface was good enough for this new SMU hardware, we were able to throw out the old code and rewrite it. Note that the register map and bit patterns changed (e.g., the old hardware uses 16-bit registers, while the new hardware uses 32-bit registers). The most significant hack was the addition of the unconvert_range() function to convert from the old SMU’s range and current parameters into a 32-bit current value.[2]

Note that we can easily accommodate new #defines and instrument state variables.

Although we chose to keep the old instrument driver interface so that our measurement layer did not require modifications, accommodating the new driver

2. This hack allowed us to use the old instrument driver interface, which prevented the changes from percolating up into the measurement layer.

allows us to see weaknesses in that old interface. For example, we could change the source_on() function to the following signature:

```
int source_on( unsigned long current);
```

Such a modification would improve the interface by using a 32-bit value and hiding driver-specific information like the required range.

In practice, it is usually not apparent where the weaknesses are in an interface until you are required to implement a new module behind that interface. Ideally, you design an interface when you have multiple modules (i.e., instrument drivers in this case) that you wish use with that interface.

8 Accommodation of Two Instruments

Our assumption that we could discard the old SMU is invalid. Although the new HPsmu_B SMU was thought to be a "superset" of the old hardware, the operation ranges of these instruments don't quiet overlap, as illustrated below. The old HPsmu_A SMU operates at a slightly higher current.[3] A new and improved measurement system must be built that can use both SMUs simultaneously.

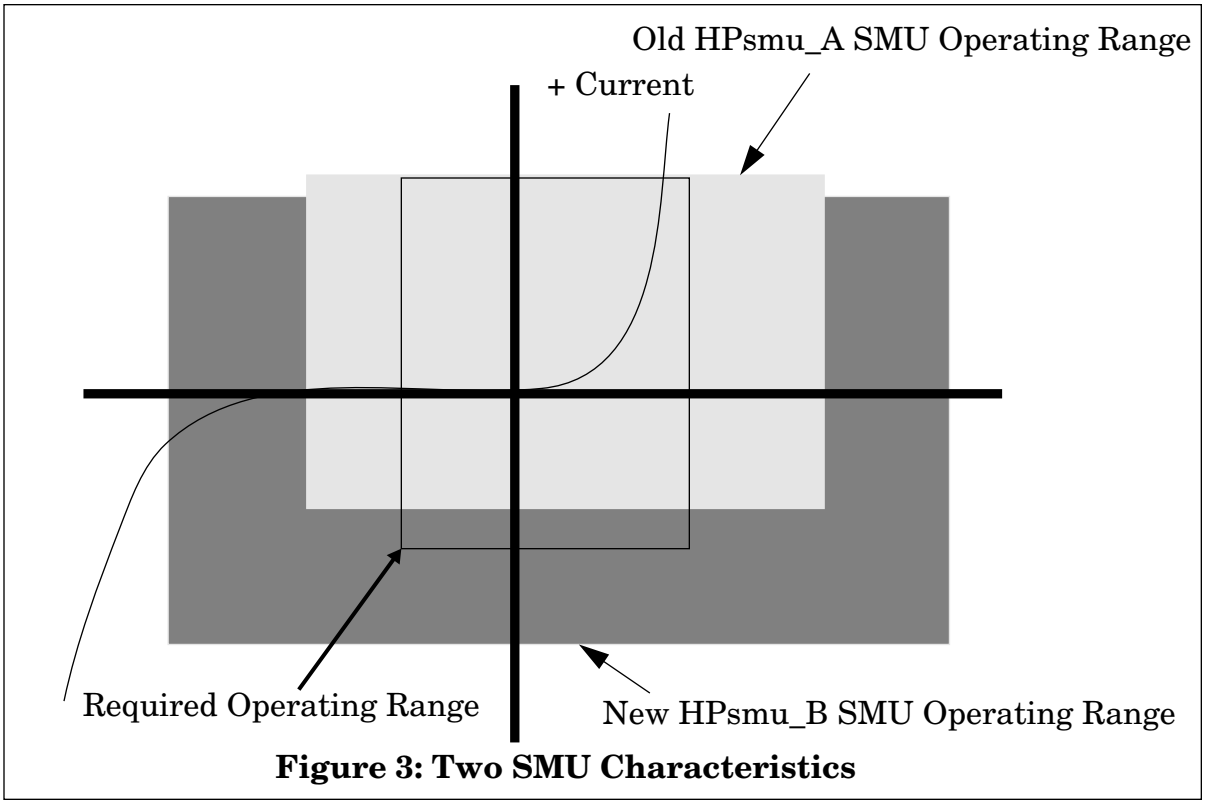

Figure 3: Two SMU Characteristics

8.1 Modifications

The biggest software change that needs to be made to the system is a mechanism to accommodate both instrument drivers simultaneously. Luckily, we can retrieve

3. The two SMUs may also differ in other parameters, like rise-time or duty cycle.

the old driver code from our source code control system. However, the following types of changes will be required.

8.1.1 Name Collisions

Because both drivers need to be linked into the application, we are forced to name the two sets of functions differently. The proposed mechanism will be to prefix the SMU model number to the function names.

For example, for the old SMU: hp_smu_a_source_on(), hp_smu_a_source_off(), ...

For the new SMU: hp_smu_b_source_on(), hp_smu_b_source_off(), ...

Note that we keep the same interface, and because each driver module is compiled separately, we don't have a problem with the #defines or the static state variables.

```
/* hp_smu_a_inst_dr.h */
/* defines for the h/w register offsets for the HPsmu_A SMU */

#define SMU_RANGE       0           /* Offset for 8-bit range register */
#define SMU_CONTROL     1           /* Offset for 8-bit control register */
#define SMU_CURRENT     2           /* Offset for 16-bit current reg */
#define SMU_VOLTAGE     4           /* Offset for 16-bit voltage reg */
...

/* driver state variables */
static unsigned long base_address;        /* Must initialize!!! */
static unsigned char control_mask;
static unsigned char range_value;         /* used for optimizations */
static unsigned short current_value;      /* used for optimizations */
...

/* function prototypes */
int hp_smu_a_source_init( int slot);
int hp_smu_a_source_on( unsigned char range, unsigned short current);
int hp_smu_a_source_not_stable();/* Report stability */
double hp_smu_a_source_voltage();/* Report source voltage */
...
```

```
/* hp_smu_b_inst_dr.h */
/* defines for the h/w register offsets for the HPsmu_B SMU */

#define SMU_CONTROL     0           /* Offset for 8-bit control register */
#define SMU_FREQ        1           /* Offset for 8-bit freq control reg */
#define SMU_CURRENT     2           /* Offset for 32-bit current reg */
#define SMU_VOLTAGE     6           /* Offset for 32-bit voltage reg */
...

/* driver state variables */
static unsigned long base_address;              /* Must initialize!!! */
static unsigned char control_mask;
```

```
static unsigned char desired_operating_freq;    /* new state var */
static unsigned long current_value;             /* used for optimizations */
...

/* function prototypes */
int hp_smu_b_source_init( int slot);
int hp_smu_b_source_on( unsigned char range, unsigned short current);
int hp_smu_b_source_not_stable();/* Report stability */
double hp_smu_b_source_voltage();/* Report source voltage */
...
```

8.1.2 Instrument Driver Selection

Somewhere in the code, we will need to select between the two equivalent driver routines. We make these types of modifications to the measurement layer:

```
/* meas_voltage.c */
#include “hp_smu_a_inst_dr.h”
#include “hp_smu_b_inst_dr.h”

/* static measurement data */
static double current_limit = 28.0;       /* use the new driver for
                                          currents less than this */

double meas_voltage( double current)
{
      ...

      if (current > current_limit) /* test range */
            {
            /* use the old driver */

            /* enable the driver */
            hp_smu_a_source_on( range, value);
            ...
            }
      else
            {
            /* use the new driver */

            /* enable the driver */
            hp_smu_b_source_on( range, value);
            ...
            }

      return (result);        /* return the voltage */
}
```

8.2 Observations

Although manageable, the measurement routines need to be modified to call the correct instrument driver routine, depending upon some criteria. In this case, we will use the new hardware whenever the measurement parameters are within its operating range. This modification is a simple "cut and paste" with edits to the function names.

Also, because we have two instruments in the system, we must call the hp_smu_a_source_init() and hp_smu_b_source_init() routines with the correct slot parameters.

9 Accommodation of Many Instruments

HP now makes five additional VXI-based SMUs (in addition to the two that we currently support). None of the seven SMUs has identical features and we foresee a future measurement system that may need to accommodate any of them. Basically, we have something like this:

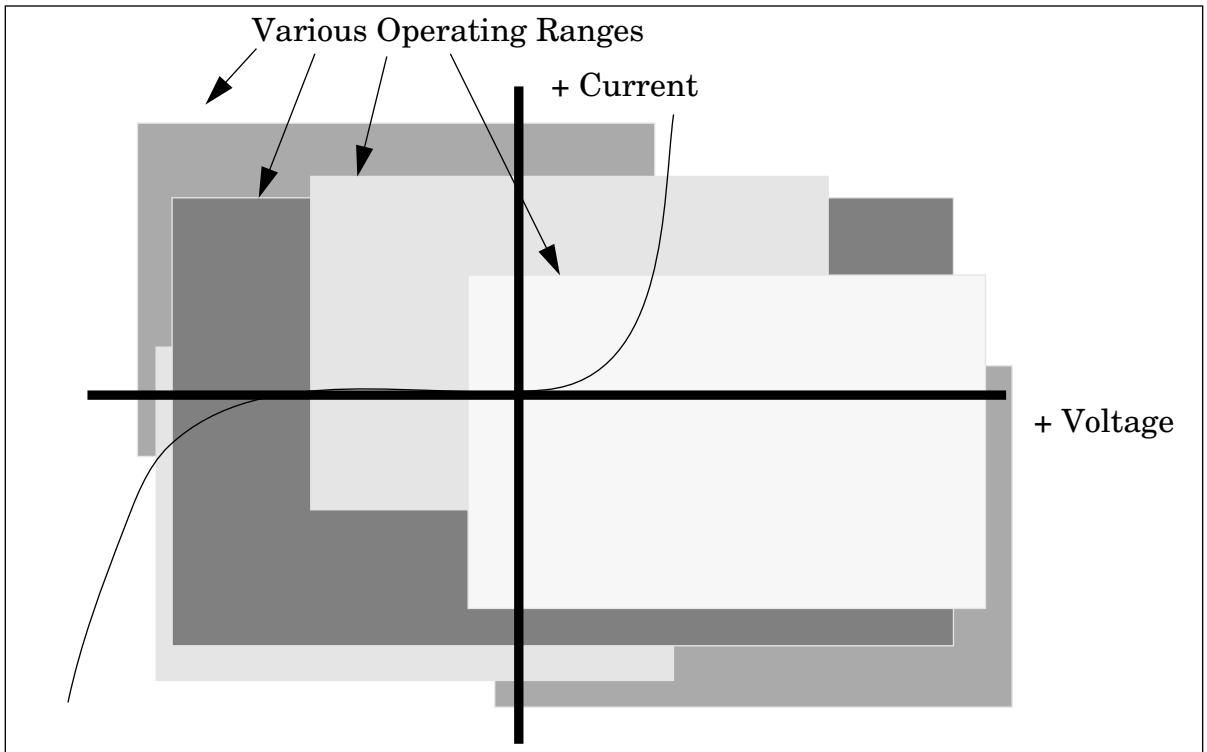

Figure 4: LED I/V Characteristics

9.1 Modifications

9.1.1 Instrument Driver Modifications

Name collisions between the instrument drivers are a big deal. We continue our strategy of prefixing instrument model numbers to the function names. We continue to manage instrument driver state using statics.

Because we can have lots of SMU instruments in the system, we must be careful to call the appropriate “source_init()” routine with the correct VXI slot parameter.

9.1.2 Measurement-Layer Modifications

The biggest problem is still calling the correct driver. Our measurement-layer functions require extensive modifications. To make the system more extensible, we choose to add a driver ID that is passed in. Also, we introduce a “DRIVER_ID” enum to keep track of which hardware is being used.

```
/* meas_voltage.h */
enum DRIVER_ID_ENUM { HPsmu_A, HPsmu_B, HPsmu_C, HPsmu_D, ...};
typedef enum DRIVER_ID_ENUM DRIVER_ID;

/* function prototypes */
double meas_voltage( DRIVER_ID driver_id, double current);
...


/* meas_voltage.c */
#include “meas_voltage.h”
#include “hp_smu_a_inst_dr.h”
#include “hp_smu_b_inst_dr.h”
#include “hp_smu_c_inst_dr.h”
...

double meas_voltage( DRIVER_ID driver_id, double current)
{
      ...

      switch (id)
            {
            case HPsmu_A:/* HPsmu_A driver */
                  /* enable the driver */
                  hp_smu_a_source_on( range, value);
                  ...
                  break;

            case HPsmu_B:/* HPsmu_B driver */
                  /* enable the driver */
                  hp_smu_b_source_on( range, value);
                  ...
                  break;

            case HPsmu_C:/* HPsmu_C driver */
                  /* enable the driver */
                  hp_smu_c_source_on( range, value);
                  ...
                  break;
            ...
            }

      return (result);        /* return the voltage */
```

```
}
```

9.1.3 Test-Layer Modifications

The main change is dealing with the driver ID parameters. Because we have chosen to make this a configuration parameter, the test_led() requires modification:

```
/* test_led.c */
#include "test_led.h"
#include "meas_voltage.h"

int test_led(
      int num,                /* input, number of currents */
      DRIVER_ID driver_ids[], /* input, array of driver id's */
      double currents[],      /* input, array of current values */
      int led_categories[],   /* input, array of led bin classifications */
      double voltage[],       /* output, measured voltages */
      double color[],         /* output, measured colors */
      double light_power[],   /* output, measured light power */
      int led_bins[])         /* output, calculated LED bins */
{
      int i, status = TRUE;

      /* loop through the desired currents */
      for (i=0; i<num; i++)
            {
            /* make fundamental measurements */
            voltage[i] = meas_voltage( driver_id[i], currents[i]);
                                       /* source on */
            ...
            }
      return (status);
}
```

9.2 Observations

The interface to the measurement layer changed to accommodate the driver IDs. This forced modifications in the test layer. Note that from the test layer's perspective, the driver_id[] values are considered opaque data that are passed around but not used in the test routine.

Most of the changes in the measurement layer are conceptually straightforward. However, in reality these kinds of changes are quite tedious and error-prone. Because there would be many measurement routines in a real system, addition of a new instrument becomes a tremendous task of adding a new "case" block in each "switch" statement.

Although the above implementation is very crude, we have built an application that can accommodate any of the supported SMUs simultaneously. Because of the

way we manage instrument state, we can have only one SMU of a given type in the system.

10 Accommodation of Multiple Instruments of the Same Type

The earlier enhancements dealt primarily with handling new hardware devices. In this section, we add code to accommodate multiple SMUs of the same type. For example, we could build a “two-headed” measurement system that must test two LEDs at the same time (one SMU per LED test fixture). Alternatively, we may be interested in improved performance and wish to exploit the parallelism that two instruments may provide.

10.1 Modifications

Accommodating multiple instruments at the same time requires modifications to our mechanism for handling instrument state. We can no longer use “static” data for instrument state because we need separate copies associated with each instrument. Passing this instrument driver state information will require modifications to most layers of our application.

10.1.1 Instrument Driver Modifications

Rather than using “static” data, we now bundle all instrument state into structs and modify the instrument driver interface to take a pointer as the first parameter. Although each instrument model has a different struct, we make a hack and pass them in via a “void *” mechanism. This allows us to keep the same interface for all instrument drivers.

To initialize this structure, the “source_init” function for each driver has been changed to return a “void *”.

Note the “cast and pray” operation to deal with the “void *” in the “source_on()” routine![4]

```
/* hp_smu_a_inst_dr.h */
...
/* driver state variables for HPsmu_A SMU */
typedef struct {
      unsigned long base_address;
      unsigned char control_mask;
      unsigned char range_value;                /* used for optimizations */
      unsigned short current_value;             /* used for optimizations */
      ...
      } HPsmu_A_Dr_State;

/* --- function prototypes --- */
void *hp_smu_a_source_init( int slot);
```

4. In general, violating the type safety mechanism of ANSI C is a bad idea. However, in this case, we don’t have a choice because we want to keep the same interface for all instrument drivers.

```
int hp_smu_a_source_on( void *void_state,
      unsigned char range, unsigned short current);
...

/* hp_smu_b_inst_dr.h */
...
/* state variables for HPsmu_B SMU */
typedef struct {
      unsigned long base_address;
      unsigned char control_mask;
      unsigned char desired_operating_freq;     /* new state var */
      unsigned long current_value;              /* used for optimizations */
      ...
      } HPsmu_B_Dr_State;

/* --- function prototypes --- */
void *hp_smu_b_source_init( int slot);
int hp_smu_b_source_on( void *void_state,
      unsigned char range, unsigned short current);
...
```

Example driver source file:

```
/* --- hp_smu_b_inst_dr.c --- */
int hp_smu_b_source_on( void *void_state,
      unsigned char range, unsigned short current)
{
      /* --- Danger! Cast & Pray operation! --- */
      HPsmu_B_Dr_State *real_state = (HPsmu_B_Dr_State *) void_state;
      ...

      /* optimization if we are at the desired conditions */
      if (c_value == real_state->current_value) return (TRUE);

      ...
      }
```

Two different “source_init()” routines are provided. It is assumed all fields in the structs are initialized properly either by use of some tool or by parsing a configuration file. In this case, we make a major hack and just hard-code the initialization code. Observe how the “slot” input parameter is used to get this instrument’s base address.

```
/* --- hp_smu_a_inst_dr.c --- */
void *hp_smu_a_source_init( int slot)
{
      /* Note: malloc of new struct instance */
      HPsmu_A_Dr_State *real_state = (HPsmu_A_Dr_State *)
            malloc( sizeof( HPsmu_A_Dr_State));

      /* fill in struct */
      real_state->base_address = VXI_get_base_address( slot);
```

```
      real_state->control_mask = ...;
      real_state->range_value = ...;
      real_state->current_value = ...;
      ...

      /* Note: return as “void *”, caller must free()!! */
      return (void *) real_state;
}

/* --- hp_smu_b_inst_dr.c --- */
void *hp_smu_b_source_init( int slot)
{
      /* Note: malloc of new struct instance */
      HPsmu_B_Dr_State *real_state = (HPsmu_B_Dr_State *)
            malloc( sizeof( HPsmu_B_Dr_State));

      /* fill in struct */
      real_state->base_address = VXI_get_base_address( slot);
      real_state->control_mask = ...;
      real_state->desired_operating_freq = ...;
      real_state->current_value = ...;
      ...

      /* Note: return as “void *”, caller must free()!! */
      return (void *) real_state;
}
```

10.1.2 Measurement-Layer Modifications

Because the signatures of all the instrument drivers have changed, we need to make appropriate modifications to the measurement layer. The biggest problem is managing the driver struct pointers that are passed in from the test layer. Note that we blindly treat all pointers as “void *”.

```
/* meas_voltage.c */
#include “meas_voltage.h”
#include “hp_smu_a_inst_dr.h”
...

double meas_voltage( void *void_state, DRIVER_ID driver_id, double current)
{
      ...

      switch (id)
            {
            case HPsmu_A:/* HPsmu_A driver */
                  /* enable the driver */
                  hp_smu_a_source_on( void_state, range, value);
            ...
```

10.1.3 Test-Layer Modifications

The biggest change to the test layer relates to the driver state structs, which are passed down to the measurement routine. In our test loop, we pull pointers out of an array and blindly pass them down.

```
/* --- test_led.c --- */
#include "test_led.h"
#include "meas_voltage.h"
...
int test_led(
      int num,                /* input, number of currents */
      DRIVER_ID driver_ids[], /* input, array of driver id's */
      void *driver_states[],  /* input, array of pointers to dr structs */
      double currents[],      /* input, array of current values */
      int led_categories[],   /* input, array of led bin classifications */
      double voltage[],       /* output, measured voltages */
      double color[],         /* output, measured colors */
      double light_power[],   /* output, measured light power */
      int led_bins[])         /* output, calculated LED bins */
{
      int i, status = TRUE;

      /* loop through the desired currents */
      for (i=0; i<num; i++)
            {
            /* make fundamental measurements */
            voltage[i] = meas_voltage( driver_states[i], driver_id[i],
                  currents[i]);     /* source on */
            ...
            }
      return (status);
}
```

The real work is accomplished in the test_init() routine, where this array is constructed. Note the calls to the "source_init()" driver routines.[5]

```
/* --- test_init.h --- */
/* global test configuration data */
const int num = 3;            /* number of currents */
DRIVER_ID driver_ids[3];      /* array of driver id's */
int slots[3];                 /* array of VXI slot numbers */
void *(driver_states[3]);     /* array of pointers to dr structs */

/* function prototype */
int test_init();


/* --- test_init.c --- */
#include "test_init.h"
```

5. Observe that a test-layer routine (e.g., test_init()) is calling driver-layer code (e.g., "source_init()" routines.

```
#include “meas_voltage.h”
#include “hp_smu_a_inst_dr.h”
#include “hp_smu_b_inst_dr.h”
#include “hp_smu_c_inst_dr.h”
#include “hp_smu_d_inst_dr.h”
...

int test_init()
{
      int i;

      /* Hack routine to initialize test configuration parameters */
      /* Assume the following test setup:
            VXI Slot          Instrument Model
            3                 HPsmu_A SMU
            4                 HPsmu_A SMU
            6                 HPsmu_B SMU
      */
      driver_ids[0] = HPsmu_A; slot[0] = 3;/* HPsmu_A SMU */
      driver_ids[1] = HPsmu_A; slot[1] = 4;/* HPsmu_A SMU */
      driver_ids[2] = HPsmu_B; slot[2] = 6;/* HPsmu_B SMU */

      for (i=0; i<num; i++)
            {
            case HPsmu_A: /* HPsmu_A SMU */
                  driver_states[i] = hp_smu_a_source_init( slot[i] );
                  break;

            case HPsmu_B: /* HPsmu_B SMU */
                  driver_states[i] = hp_smu_b_source_init( slot[i] );
                  break;
      ...
```

10.2 Observations

The biggest thing that changed when we needed to accommodate more than one instrument of the same type is our former use of static state variables. Our approach was to build a unique struct for each instrument type and to “malloc” a copy of that struct for each instrument of that type in the system.

In addition to passing pointers to these structs between the software layers, test_init() and source_init() routines were introduced to deal with initialization details. Also, in order to keep the same instrument driver interface for all instruments, we were forced to pass the struct pointers around as “void *”. Unfortunately, we had to violate the type system with a few “cast and pray” operations.

Keeping things in sync became a mild headache. For example, the instrument IDs need to match just what struct was “malloc-ed” and placed in the driver_states[] array. Care was needed to maintain agreement with the case enums in the meas_voltage() and test_init() functions.

The above implementation is still very crude (although growing in complexity). However, we have built an application that can accommodate any number of the supported SMUs simultaneously. We can reconfigure our measurement system very easily to use different hardware. In a real system, the hard-coded initialization in test_init() and source_init() routines would be modified to read necessary parameters from a configuration file or via some test integration tool.

11 Adoption of Object-Oriented Techniques

In this section, we introduce object-oriented techniques to clean up the example program. However, we do not change the language and still code in ANSI C.

11.1 Standard Interfaces and Jump Tables

As should be apparent from the earlier examples, we have consistently maintained the same interface for all instrument drivers. The only thing that differs is the actual function names (which contain the instrument driver model number as a prefix).

We formalize this approach and lock down this interface with a jump table:

```
/* inst_dr_table.h */

/* Define Jump Table struct */
typedef struct {
      void *(source_init)( int slot);           /* Init struct */
      int (*source_on)( void *state, unsigned char range,
            unsigned short current);            /* Source on */
      int (*source_not_stable)(void *state);    /* Report stability */
      double (*source_voltage)(void *state);    /* Report source voltage */
      ...
      } INST_DR_TABLE;
```

We will need an instance of the above jump table for each type of instrument in the system. Note that we can share this table when we have multiple instruments of the same type. Also, we need to initialize the pointers in the table with the appropriate driver function names. For each driver, we will introduce a "table_init()" routine. For example, for the HPsmu_A SMU driver, we have the following routine:

```
/* hp_smu_a_inst_dr.c */
#include "inst_dr_table.h"

INST_DR_TABLE *hp_smu_a_table_init()
{
      INST_DR_TABLE *my_table = (INST_DR_TABLE *)
            malloc( sizeof( INST_DR_TABLE));

      my_table->source_init = hp_smu_a_source_init;
      my_table->source_on = hp_smu_a_source_on;
```

```
        my_table->source_not_stable = hp_smu_a_source_not_stable;
        my_table->source_voltage = hp_smu_a_source_voltage;
        ...

        return (my_table);
}
```

11.2 Instrument Driver Structure

For each instrument driver, we define a struct that contains a pointer to the jump table and a pointer to the driver state. We will have an instance of this struct for each instrument in the system (because we need a separate copy of the driver state). However, we can share the jump tables between instances of the same hardware type.

```
/* --- inst_dr.h --- */
typedef struct {
      INST_DR_TABLE *table;          /* pointer to jump table */
      void *driver_state;            /* "void *" pointer to real struct */
      } SMU_INST_DR;
```

11.2.1 Using the Instrument Driver Structure

Our measurement layer becomes greatly simplified because of the jump tables. The meas_voltage() function becomes:

```
/* meas_voltage.c */
#include "inst_dr.h"          /* Note: no include files for each driver */
#include "meas_voltage.h"

double meas_voltage( SMU_INST_DR *inst_dr, double current)
{
      /* local variable to simplify access */
      INST_DR_TABLE *table = inst_dr->table;
      void *driver_state = inst_dr->driver_state;

      /* enable the driver */
      table->source_on( driver_state, range, value);

      /* wait for the driver to stablize, loop until stable */
      while ( table->source_not_stable( driver_state) );

      /* read the voltage and convert to double */
      v_value = table->source_voltage( driver_state);
            ...
      }
```

In the above code, observe the following points:

- Only the generic "inst_dr.h" include file is needed by the measurement layer. We do not need to include the specific drivers for each instrument. This speeds compiles because we don't force the inclusion of each driver's .h file.

- All calls to the driver layer are made via the jump table. The “switch” statement is no longer needed in order to select which driver routine to invoke.
- The meas_voltage code is fully reusable when new instrument hardware is added to the system. Assuming the interface is stable (i.e.: the jump table doesn’t change), this code does not need to be edited or recompiled just to add a new instrument driver.
- Local variables are used to simplify access and to improve readability of the code.

11.2.2 Initialization of the Instrument Driver Structure

The key becomes the initialization of the instrument driver structures. We make the following modifications to the test_init() routine:

```
/* --- test_init.h --- */
/* global test configuration data */
const int num = 3;      /* number of currents */
SMU_INST_DR inst_dr[3]; /* array of instrument drivers */

/* function prototype */
int test_init();


/* --- test_init.c --- */
#include “test_init.h”
#include “meas_voltage.h”
#include “inst_dr.h”          /* generic driver include file */

/* specific instrument driver include files */
#include “hp_smu_a_inst_dr.h”
#include “hp_smu_b_inst_dr.h”
#include “hp_smu_c_inst_dr.h”
#include “hp_smu_d_inst_dr.h”
...

int test_init()
{
      int i;

      /* Hack routine to initialize test configuration parameters */

      /* Initialize each jump table */
      /* Note that we make one instance per type of inst H/W */
      INST_DR_TABLE *hp_smu_a_table = hp_smu_a_table_init();
      INST_DR_TABLE *hp_smu_b_table = hp_smu_b_table_init();

      /* Assume test setup with two HPsmu_A SMUs and one HPsmu_B SMU
            in VXI slots 3, 4, and 6 */
      inst_dr[0].table = hp_smu_a_table;
      inst_dr[0].driver_state = hp_smu_a_source_init( 3);

      inst_dr[1].table = hp_smu_a_table;  /* Note: reuse table */
      inst_dr[1].driver_state = hp_smu_a_source_init( 4 );
```

```
        inst_dr[2].table = hp_smu_b_table;
        inst_dr[2].driver_state = hp_smu_b_source_init( 6 );
        ...
}
```

Observe the following points:

- In reality, we would probably read configuration information from a file or perform the initialization via a tool; the above mechanism hints at what is required.
- An instance of the jump table is made for each instrument type. In this case, we make two jump tables.
- An instance of the appropriate driver state struct is made for each instrument in the system. In this case, we make three driver state structs.
- We must be very careful to keep the correct jump table bundled with the proper driver state struct. We manage this association in the inst_dr array.

11.2.3 Driver Inheritance

With the above architecture and mechanism to initialize the jump tables, we can also support a manual form of object inheritance. Imagine the need to add a new instrument to the system that differs only slightly from an existing instrument. For example, assume we are adding an HPsmu_C SMU that is just a more recent and slightly improved version of the HPsmu_B SMU.

To illustrate the process, we assume that only the hp_smu_b_source_on() routine needs to be modified and that all of the other "B" driver routines will work fine with the new hardware. Therefore, we must provide a new hp_smu_c_source_on() and must initialize the "C" state struct and jump table as follows:

```
/* hp_smu_c_inst_dr.h */
#include "hp_smu_b_inst_dr.h"       /* Note include the parent's defs */
...
/* state variables for HPsmu_C SMU */
typedef struct {
        /* A copy of the "B" H/W state variables */
        HPsmu_B_Dr_State b_state;           /* This MUST BE FIRST */

        /* Add new state that is unique to the "C" H/W */
        unsigned char rise_time;
        unsigned char turn_on_delay;
        ...
        } HPsmu_C_Dr_State;

/* --- function prototypes --- */
void *hp_smu_c_source_init( int slot);
int hp_smu_c_source_on( void *void_state,
        unsigned char range, unsigned short current);
...
```

```
/* hp_smu_c_inst_dr.c */
#include “inst_dr_table.h”
#include “hp_smu_c_inst_dr.h”

int hp_smu_c_source_on( void *void_state,
      unsigned char range, unsigned short current)
{
      /* --- Danger! Cast & Pray operation! --- */
      HPsmu_C_Dr_State *real_state = (HPsmu_C_Dr_State *) void_state;
      ...
}

void *hp_smu_c_source_init( int slot )
{
      /* Note: malloc of new struct instance */
      HPsmu_C_Dr_State *real_state = (HPsmu_C_Dr_State *)
            malloc( sizeof( HPsmu_C_Dr_State));

      /* Get an initialized B struct */
      HPsmu_B_Dr_State *b_state = hp_smu_b_source_init( slot );

      /* Copy the B H/W state struct across */
      real_state->b_state = *b_state;

      /* fill in unique state for the C H/W */
      real_state->rise_time = ...;
      real_state->turn_on_delay = ...;
      ...

      /* Note: return as “void *”, caller must free()!! */
      return (void *) real_state;
}

INST_DR_TABLE *hp_smu_c_table_init()
{
      /* Get a jump table for the HPsmu_B H/W */
      INST_DR_TABLE *my_table = hp_smu_b_table_init();

      /* Change the source_init and source_on function pointers */
      my_table->source_init = hp_smu_c_source_init;
      my_table->source_on = hp_smu_c_source_on;
      return (my_table);
}
```

The hp_smu_c_table_init() routine is fairly straightforward. We get a jump table that is appropriate for the “B” hardware and then modify the source_init and source_on pointers appropriately. Observe that this pointer replacement to the “C” routines is the real mechanism behind inheritance.

The hp_smu_c_source_init() routine is very tricky. Recall that the measurement layer will blindly pass the state struct down to the instrument driver routines as a "void *". Because we are reusing most of the "B" functions, we need to make sure that this address points to an HPsmu_B_Dr_State struct. We accomplish this by placing an HPsmu_B_Dr_State struct inside the HPsmu_C_Dr_State struct as the first element. Note that it is critical that this be the "first" element! Any state unique to the "C" hardware must be added after this element.

Note also that the above scheme gives us a form of object inheritance. We are able to reuse most of the "B" instrument driver routines and add only new code when needed. Observe that we reuse the code and do not leverage it. For example, if improvements to the "B" code are made (i.e., imagine a bug is found and fixed in the hp_smu_b_source_not_stable() routine), we inherit that change directly. We do not need to make modifications to the "C" source code, although our include file dependences will probably force a recompilation.

11.3 Observations

In the above code fragments, we formalized the instrument driver interface by adding jump tables. We continued to manage driver state as structs (i.e., no statics!) and bound the two together in a new SMU_INST_DR struct.

By far the biggest benefit is realized in our measurement layer. We now have an architecture that will allow full measurement code reuse. In fact, we do not even require recompilation of the measurement layer when new instrument drivers are added to the system! As before, we can accommodate any instrument driver that obeys the instrument driver interface and we can support multiple instruments of the same type (i.e., the driver code is shared but each instrument has a unique copy of the driver state struct).

The measurement code assumes that it can access a generic driver via the provided jump table. In reality, a generic driver will never exist. For example, we don't have a generic source_on() function, but only specific ones like hp_smu_a_source_on(), or hp_smu_b_source_on(). However, the measurement routines think that we do! The jump table allows us to manage this illusion.

We also have illustrated a manual form of object inheritance. Although initialization is tricky, we can re-use code and make modifications only to those routines that require it. Note that this is not code leverage, but is true code reuse!

The key to this architecture is in the initialization of the jump tables and driver state structs. We handle this in the test_init(), table_init(), and source_init() routines.

Although we have adopted many object-oriented concepts (like abstract interfaces, encapsulation, and object inheritance), we have done this in straight ANSI C. However, it should be clear that a great deal of detail must be provided in the form of initialization routines in order to make this scheme work.

12 Rewriting the Code in C++

In this section, we rewrite the example in C++. As before, our focus is limited only to the instrument driver layer where each instrument will be represented by a C++ class. In reality, these concepts could be easily extended to the measurement and test layers.

12.1 Abstract Instrument Driver Class

Our jump table becomes an abstract class in C++. In this case, none of the methods (except the constructor) of the abstract class are real.[6] The main purpose for this class is to lock in the interface.

In addition, we use the abstract class as a convenient place to hold state variables that are common to all SMU drivers. In the abstract class's constructor, we initialize this common state.

```
// inst_dr.hh
class SMU_INST_DR {
      public:
            SMU_INST_DR( int slot );                  // Constructor
            virtual ~SMU_INST_DR();                   // Destructor

            virtual int source_on( unsigned char range,
                  unsigned short current) = 0;        // Source on
            virtual int source_not_stable() = 0;      // Report stability
            virtual double source_voltage() = 0;      // Report voltage
            ...
      protected:
            // Any common state variables or methods should go here
            unsigned int base_address;
            unsigned char control_mask;
};

// inst_dr.cc
#include "inst_dr.hh"

// Constructor
SMU_INST_DR::SMU_INST_DR{ int slot)
{
      // set defaults for common state variables
      base_address = VXI_get_base_address( slot);
      control_mask = ???;
}
```

Observe the following points:

- The methods look very similar to the functions in the above jump table. However, the driver state struct is no longer a visible part of the interface.

6. A C++ method is very similar to a C function, but is specific to the given class.

- The “virtual” notation means that this method can be modified by subclassing and overriding. In essence, a slot is defined in the hidden jump table (e.g., C++ vtable).
- The “= 0” notation means that the method is “pure virtual.” In other words, a slot is created in the hidden jump table but not filled in. Also, the method will not be supplied as part of the implementation (i.e., it will be missing from the inst_dr.cc source file).
- The source_init() routine is no longer needed. Instead, a “constructor” is used.
- The “destructor” is used when the instance is destroyed. Usually, it will be automatically called.
- Because this is primarily a “pure abstract” class, we will have only the constructor in the inst_dr.cc source file.
- Because the state variables base_address and control_mask are shared by all instrument drivers, we include them as part of the abstract interface as protected data. Because they are protected, users of the abstract class can not modify their contents. Note the common initialization of the base_address variable from the constructor’s slot parameter.

12.2 Real Instrument Driver Class

For each instrument type in the system, we will write a C++ class that contains both methods and state. In object-oriented jargon, this is called “encapsulation.” For example, the HPsmu_A SMU would be controlled by the following class:

```
// hp_smu_a_inst_dr.hh
#include “inst_dr.hh”

// defines for the h/w register offsets for the HPsmu_A SMU

#define SMU_RANGE       0           // Offset for 8-bit range register
#define SMU_CONTROL     1           // Offset for 8-bit control register
#define SMU_CURRENT     2           // Offset for 16-bit current reg
#define SMU_VOLTAGE     4           // Offset for 16-bit voltage reg
...

class HPsmu_A_INST_DR : public SMU_INST_DR {
      public:
            HPsmu_A_INST_DR( int slot );              // Constructor
            virtual ~HPsmu_A_INST_DR();               // Destructor

            virtual int source_on( unsigned char range,
                  unsigned short current);            // Source on
            virtual int source_not_stable();          // Report stability
            virtual double source_voltage();          // Report voltage
            ...

      protected:
            // Add new state that is unique to the “A” H/W
            unsigned char range_value;          // used for optimizations
```

```
            unsigned short current_value;       // used for optimizations
      ...
};
```

Note that this class inherits from the abstract interface (see the "public SMU_INST_DR" in the first line of the class), and that the former driver state is now part of the class (in the "protected" section). Since this is protected data, users of this class (e.g., the measurement layer) will not have any knowledge of these variables.

A sketch of a partial implementation would be as follows:

```
// hp_smu_a_inst_dr.cc
#include "hp_smu_a_inst_dr.hh"

HPsmu_A_INST_DR::HPsmu_A_INST_DR( int slot) : SMU_INST_DR( slot)//
Constructor
{
      // Initialize the state to reasonable defaults
      range_value = ...;
      current_value = ...;
}

~HPsmu_A_INST_DR::HPsmu_A_INST_DR()       // Destructor
{
      source_on( 0, 0);             // make sure the H/W is off
      // any additional clean-up
      ...
}


int HPsmu_A_INST_DR::source_on( unsigned char range,
      unsigned short current)             // Source on
{
      // optimization if we are at the desired conditions
      if (range == range_value && current == current_value)
            return (TRUE);

      // disable the source before we change the values
      control_mask &= 0xf7;   // clear the enable bit
      VXI_write_byte( base_address+SMU_CONTROL, control_mask);
      ...
}

int HPsmu_A_INST_DR::source_not_stable()  // Report stability
{
...
}

double HPsmu_A_INST_DR::source_voltage()  // Report voltage
{
...
```

```
}
```

Observe the following points:

- The initialization code is greatly simplified and reduced. The C++ compiler manages the jump table automatically. The only work for us is the setting of default values inside the “constructor.. In this case, we set only defaults for state variables that are unique to this subclass. Note that the abstract base class constructor will initialize the common state.
- Because the “destructor” will be automatically called when an instance is destroyed, we place our shutdown code there. In this case, we just make sure that the hardware is disabled by setting the current to 0. Note that the destructor call invokes the source_on() method.
- Accessing the driver state variables is through simple and obvious notation. We no longer see or manage the old driver state structs.

12.3 Code Inheritance

To illustrate specialization and code reuse, here is how we accommodate the HPsmu_C hardware. We assume that a HPsmu_B_INST_DR class exists and is subclassed from the abstract driver class as follows:

```
// hp_smu_b_inst_dr.hh
#include “inst_dr.hh”
...
class HPsmu_B_INST_DR : public SMU_INST_DR {
      public:
            HPsmu_B_INST_DR( int slot );              // Constructor
            virtual ~HPsmu_B_INST_DR();               // Destructor

            virtual int source_on( unsigned char range,
                  unsigned short current);            // Source on
            virtual int source_not_stable();          // Report stability
            virtual double source_voltage();          // Report voltage
            ...

      protected:
            // Add new state that is unique to the “B” H/W
            unsigned char desired_operating_freq;
            unsigned long current_value;
      ...
};
```

To accommodate the “C” hardware, we subclass from the “B” driver class and override the source_on() method:

```
// hp_smu_c_inst_dr.hh
#include “hp_smu_b_inst_dr.hh”
...
class HPsmu_C_INST_DR : public HPsmu_B_INST_DR {
      public:
```

```
            HPsmu_C_INST_DR( int slot );              // Constructor
            virtual ~HPsmu_C_INST_DR();               // Destructor

            virtual int source_on( unsigned char range,
                  unsigned short current);            // Source on
      protected:
            // Add new state that is unique to the "C" H/W
            unsigned char rise_time;
            unsigned char turn_on_delay;
            ...
};
```

Although the implementation of the "B" and "C" driver methods is not provided, it is very straightforward. By far the biggest advantage is that we don't need to manage the old driver state structs or jump tables. The C++ compiler takes care of those nagging details.

Also, for the "C" class, all constructors will be called in this order (from abstract to most specialized): (1) SMU_INST_DR, (2) HPsmu_B_INST_DR, and (3) HPsmu_C_INST_DR. The result will be automatic initialization of all state variables.

12.4 Initialization Details

Our old test_init() routine is modified to deal with the C++ classes as follows:

```
// --- test_init.hh ---
// global test configuration data
const int num = 3;      // number of currents
SMU_INST_DR *inst_dr[3];// array of pointers to instrument drivers

// function prototype
int test_init();


// --- test_init.cc ---
#include "test_init.hh"
#include "meas_voltage.hh"
#include "inst_dr.hh"         // generic driver include file

// specific instrument driver include files
#include "hp_smu_a_inst_dr.hh"
#include "hp_smu_b_inst_dr.hh"
#include "hp_smu_c_inst_dr.hh"
#include "hp_smu_d_inst_dr.hh"
...

int test_init()
{
      // Hack routine to initialize test configuration parameters

      // Assume test setup with two HPsmu_A SMUs and one HPsmu_B SMU
      // in VXI slots 3, 4, and 6
      // Create an instance of the appropriate driver class for
```

```
      // each H/W. Note that the "constructors" are automatically
      // called. Each instance pointer is saved in an array.
      inst_dr[0] = new hp_smu_a_inst_dr( 3 );
      inst_dr[1] = new hp_smu_a_inst_dr( 4 );
      inst_dr[2] = new hp_smu_b_inst_dr( 6 );
      ...
}
```

Observe that the only real work in the test_init() routine is a series of "new" operations. The C++ new operator will automatically create enough space for the instance and then call the appropriate constructors.[7]

Although not shown, a configuration file or tool would still be used to guide this instantiation process. For now, we just hard-code three calls to new. Also, the current code is very simple and a mechanism would be needed that could modify state parameters in each instance after the constructors are called.

12.5 Using the C++ Instrument Driver Classes

Our measurement routine becomes simpler, as follows:

```
// meas_voltage.cc
#include "inst_dr.hh"         // Note: no include files for each driver
#include "meas_voltage.hh"

double meas_voltage( SMU_INST_DR *inst_dr, double current)
{
      // enable the driver
      inst_dr->source_on( range, value);

      // wait for the driver to stablize, loop until stable
      while ( inst_dr->source_not_stable( ) );

      // read the voltage and convert to double
      v_value = inst_dr->source_voltage( );
            ...
      }
```

The following benefits are realized:

- The measurement code sees only the abstract instrument driver class. None of the driver subclasses are required.
- A pointer to the instrument driver is provided as the first parameter. Although this is a pointer to an abstract class, in reality it points to a real driver instance.
- The driver state struct is gone and the driver state variables are now completely hidden from the measurement layer.[8]

---

7. This is analogous to the old malloc calls in the source_init() routines.
8. Recall that the state variables are now protected data of the class. Consequently, they are not visible to users of the class like this meas_voltage routine.

- The above code would work with any subclass of the SMU_INST_DR abstract class. More importantly, the C++ compiler will handle most of the details in setting up and managing hidden jump tables and state variables.

13 Conclusions

If you are a C programmer, hopefully you are convinced that there are many benefits to adopting some object-oriented techniques in your code. The following concepts can be and have been used without changing to an object-oriented language:

- Organize routines into standard interfaces and manage access with jump tables.
- Discourage the use of static state variables. They prohibit multiple instances of the same object.
- Bundle necessary state variables and organize them into state structs.
- Encapsulate both state and routines by passing the state structs as the first parameter to all calls on an interface. From routines outside of the interface, treat these structs as opaque data structures.
- Organize the state struct and appropriate jump table together into an interface struct.
- When supporting multiple objects of the same type, observe that jump tables can be shared but state struct instances should never be shared.
- Provide initialization routines to set up jump tables and state structs. These routines should malloc required storage. Also, ensure that these routines are called at the appropriate times in your application.
- Provide clean-up routines to free storage like jump tables and state structs. Ensure that they are called at the appropriate times.
- With the use of appropriate tricks in initializing jump tables and state structs, a simple form of inheritance can be realized.
- Design an appropriate system-level initialization mechanism that will allow you to easily modify which state structs are created, what default values are assigned, and how these pointers get passed around the system.

Hopefully it is clear from this document that switching to an object-oriented language like C++, although not required, has numerous benefits. From an architecture perspective, the following points are suggested from the earlier examples (but really come from much C++ application experience):

- Build your software architecture so that there is a rich hierarchy of classes with clearly defined abstract interfaces.
- Organize your classes into the following groups:
  1. Application classes that model the top-level objects in your system. For example, tests, measurements, instruments, auto-handlers. Classes in this group may be reusable in similar applications.

2. Infrastructure classes that facilitate the operation of your application-level classes. For example, classes that manage initialization support, communications, O/S services, U/I support. Classes in this group should be reusable across most applications in your domain.
3. Utility classes that provide common programming services like smart strings, smart arrays, hash tables. You should consider purchasing these classes (e.g., Booch, Microsoft Foundation Classes). Classes in this group are very general and should be reusable across a wide range of applications.

- Exploit "is a," "has a," and "uses a" relationships between classes in your system.

  "is a" — This is the classic inheritance relationship. For example, an HPsmu_C_INST_DR class really is an HPsmu_B_INST_DR and is also a SMU_INST_DR.

  "has a" — This is when your class completely owns another class instance as a state variable.

  For example, your HPsmu_C_INST_DR class may use a smart array utility class as part of the implementation. In the HPsmu_C_INST_DR constructor, the array should be created and initialized. In the destructor, the array should be deleted.

  "uses a" — This is when your class uses another object in the system but does not own it. Typically, your class has a pointer to the "used" instance as a state variable.

  The "uses a" relationship is most powerful when the pointer is to an abstract interface class. For example, a MEAS class would have this state variable: "SMU_INST_DR *driver".

  Note that this pointer must be initialized and typically will point to a real object like an HPsmu_B_INST_DR.

- Require that all class constructors initialize state variables to reasonable defaults.
- Provide a mechanism to instantiate classes and to make assignments to all the "uses a" pointers. This is a dynamic link operation between previously constructed classes.
- Provide a generic accessor mechanism by which state variable can be changed from defaults to specific values that are required in the running application.
- A mechanism will be needed that can initialize the application. Although the following steps can certainly be hard-coded, it is recommended that an infrastructure class be developed that performs them by reading a database or parsing an initialization file:

  1. Create required instances of the main application-level classes.[9]

9. Note that the constructors for the application-level classes are executed. Assuming that "has a" class relationships are exploited, the constructors for such utility classes will also be executed.

2. Use the generic accessor mechanism to change select state variables to values that are required for the application.
3. Use the dynamic link mechanism to assign all “uses a” pointer.
4. Verify that the initialization was complete and that all instances are in a runable state.
5. Launch the main execution sequence for your application.

From an implementation perspective, C++ simplifies the implementation as follows:

- Most of the initialization code can and should be handled by the C++ compiler. Without compiler support, a great deal of time building jump tables and state structs will be required. Such code is also very error-prone.
- The hack of passing state structs around the system as “void *” is completely removed.
- Many forms of inheritance can be managed by the C++ compiler. For example, multiple inheritance in many forms (e.g., virtual public inheritance) are supported.
- C++ is a strongly typed language and builds upon ANSI C’s optional use of function prototypes. As a word to the uninitiated, pay close attention to all C++ warnings!
- Through proper use of constructors and destructors, many of the nasty problems with C memory leaks and accessing uninitialized data can be prevented.
- Although not illustrated in this document, C++ provides the ability to define operators for a class. This can be effective for utility classes like strings, where the ‘+’ operator can be used to perform concatenation, or for numeric classes like “complex,” where the math operators ‘+’, ‘-’, ‘*’, ‘/’ can be defined to perform complex mathematics.