HP 4062UX Semiconductor Process Control System (For S700)

Programming Guide for C Library

HP Part No. 04062-90355
Printed in Japan December 1994

Edition 1
E1294

Product Warranty

This Hewlett-Packard system product is warranted against defects in material and workmanship for a period of one year from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country.

HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modifications or misuse, operation outside of the environment specifications for the products, or improper site preparation or maintenance.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED. HEWLETT-PACKARD SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OR MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Exclusive Remedies

THE REMEDIES PROVIDED HEREIN ARE BUYER'S SOLE AND EXCLUSIVE REMEDIES. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER BASED ON CONTRACT, TORT, OR ANY OTHER LEGAL THEORY.

Assistance

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

Certification

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. HP further certifies that its calibration measurements are traceable to the National Institute of Standards and Technology (NIST), to the extent allowed by the Institute's calibration facility, and to the calibration facilities of other International Standards Organization members.

Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software revision code printed before the date indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

Edition 1 December 1994 For revision X.07.0x

Safety Summary

The following general safety precautions must be observed during all phases of operation, service, and repair of this instrument. Failure to comply with these precautions or with specific warnings elsewhere in this manual violates safety standards of design, manufacture, and intended use of the instrument. Hewlett-Packard Company assumes no liability for the customer's failure to comply with these requirements.

GROUND THE INSTRUMENT

To minimize shock hazard, the instrument chassis and cabinet must be connected to an electrical ground. The power terminal and the power cable must meet International Electrotechnical Commission (IEC) safety standards.

DO NOT OPERATE IN AN EXPLOSIVE ATMOSPHERE

Do not operate the instrument in the presence of flammable gases or fumes. Operation of any electrical instrument in such an environment constitutes a definite safety hazard.

KEEP AWAY FROM LIVE CIRCUITS

Operation personnel must not remove instrument covers. Component replacement and internal adjustments must be made by qualified maintenance personnel. Do not replace components with power cable connected. Under certain conditions, dangerous voltages may exist even with the power cable removed. To avoid injuries, always disconnect the power and discharge circuits before touching them.

DO NOT SERVICE OR ADJUST ALONE

Do not attempt internal service or adjustment unless another person, capable of rendering first aid and resuscitation, is present.

DO NOT SUBSTITUTE PARTS OR MODIFY INSTRUMENT

Because of the danger of introducing additional hazards, do not install substitute parts or perform any unauthorized modification to the instrument. Return the instrument to a Hewlett-Packard Sales and Service Office for services and repair to ensure that safety features are maintained.

DANGEROUS PROCEDURE WARNINGS

Warnings, such as the example below, precede potentially dangerous procedures throughout this manual. Instructions contained in the warnings must be followed.

Warning

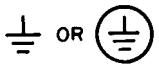
DANGEROUS VOLTAGES, CAPABLE OF CAUSING DEATH, ARE PRESENT IN THIS INSTRUMENT. USE EXTREME CAUTION WHEN HANDLING, TESTING, AND ADJUSTING.

Safety Symbols

The general definitions of safety symbols used on equipment or in manuals are listed below.

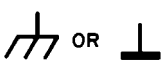
Instruction manual symbol: the product will be marked with this symbol when it is necessary for the user to refer to the instruction manual in order to protect against damage to the instrument.

Indicates dangerous voltage (terminals fed from the interior by voltage exceeding 1000 volts must be so marked).



Protective conductor terminal. For protection against electrical shock in case of a fault. Used with field wiring terminals to indicate the terminal which must be connected to ground before operating equipment.

Low-noise or noiseless, clean ground (earth) terminal. Used for a signal common, as well as providing protection against electrical shock in case of a fault. A terminal marked with this symbol must be connected to ground in the manner described in the installation (operation) manual, and before operating the equipment.



Frame or chassis terminal. A connection to the frame (chassis) of the equipment which normally includes all exposed metal structures.

Alternating current (power line).

Direct current (power line).

Alternating or direct current (power line).

Warning 

The warning sign denotes a hazard. It calls attention to a procedure, practice, condition or the like, which, if not correctly performed or adhered to, could result in injury or death to personnel.

Caution 

The caution sign denotes a hazard. It calls attention to an operating procedure, practice, condition or the like, which, if not correctly performed or adhered to, could result in damage to or destruction of part or all of the product.

Note 

The note sign denotes important information. It calls attention to a procedure, practice, condition or the like, which is essential to highlight.

Preface

Purpose

This manual is intended for the C programming environment for the HP 4062UX system installed in series 700 computers, and contains the information you need to program your HP 4062UX in C programming language for performing measurements.

Audience

This manual is intended for system programmers of the HP 4062UX system, who are familiar with C programming in an HP-UX environment.

Contents

Chapter 1, "General Information," describes test program development, hardware configuration, software concepts, the power-on procedure, and HP-UX environment settings.

Chapter 2, "Compilation and Execution," provides how to compile and execute the test program, and information on the startup command PCSSTART.

Chapter 3, "Measurement Programming," explains Test Instruction Set (TIS), which is a set of functions to control the HP 4062UX system, and describes how to program with them.

Chapter 4, "Program Debugging," provides information on debugging capabilities furnished with the HP 4062UX system.

Chapter 5, "Prober Control," provides specific wafer prober control information for using HP-IB compatible wafer probers.

Chapter 6, "Wafer Probing and Data Logging," explains wafer probing and data logging software and how to use it.

Appendix A, "Customizing Your Prober Driver," explains how to modify or create prober drivers.

Contents

3. Measurement Programming

Figures

Tables

1

General Information

This manual provides information for the C programming environment of the HP 4062UX system (Option 034). The C programming environment of the HP 4062UX system lets you develop test programs in the C programming environment of HP-UX.

For hardware information and operating theory, see the *System Information* manual. For the Virtual Front Panel (VFP), Probing Pattern Generator (PPG), Wafer Mapping (MAP) program, and other standalone utilities, which run in the HP BASIC/UX environment, see the *System Operation* manual. For the organization of this manual, see the "Preface" at the beginning of this manual.

This chapter provides basic information on the C programming environment of the HP 4062UX system. The following topics are included in this chapter:

- Test program development
- Hardware configuration
- Software concepts
- Parametric test session
- Turning the HP 4062UX system on and off
- HP-UX environment settings

Test Program Development

When using the HP 4062UX system, you are released from connecting several instruments, operating the instruments, reading the measured data, and creating several graphs. Once you program the measurement sequence, you can measure test devices automatically or semiautomatically. The following provides basic information on test program development.

In the HP-UX, C and HP BASIC/UX programming environments are supported for the HP 4062UX system. You can use the C programming language to create a program controlling the HP 4062UX system. For the HP BASIC/UX environments, see the *System Operation* manual.

The test programs you create control system instruments, measure the devices under test (DUTs), and process the measured data, automatically and fast. Figure 1-1 shows a typical development flow chart of the test programs.

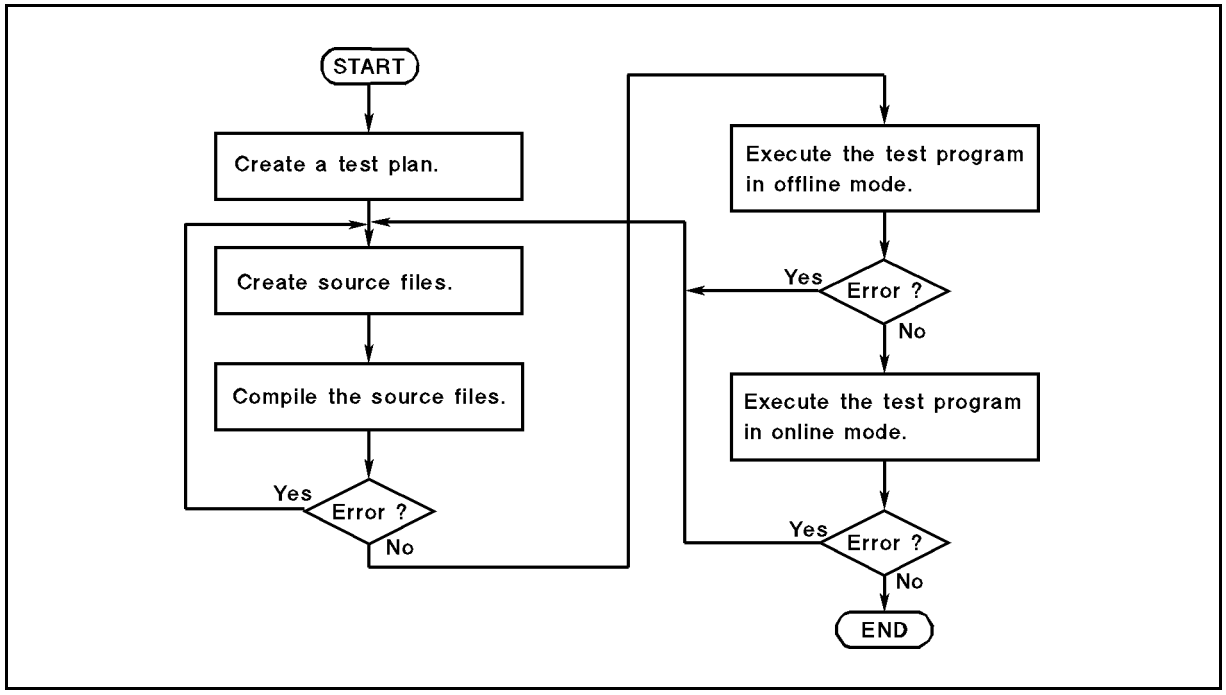


Figure 1-1. Typical Development Flow Chart of Test Programs

In the test plan, the following items are defined:

- What devices are measured? Discrete, packaged, on-wafer, and so forth.
- What characteristics are measured? h_{FE}, Vth, sheet resistance, and so forth.
- What measurement method is used to measure the characteristic? Spot measurement, staircase sweep measurement, and so forth.
- What are reported? Scattergram, histogram, and so forth.
- What database is created?

The definitions above are only examples. You must modify or create the definitions for your purpose, then create the test program to suit your needs.

Hardware configuration

The standard HP 4062UX system consists of three basic subsystems, totaling four separate instruments (not including the system controller). All subsystems are controlled via the Hewlett-Packard Interface Bus (HP-IB) by the HP 9000 Series 700 computer (system controller). The system controller is equipped with the HP-UX operating system.

The following abbreviations are used to identify the subsystems and instruments of the HP 4062 system.

- Switching Matrix Subsystem (SMS)

  The SMS consists of the following two instruments:

  HP 4084B Switching Matrix Controller (SWC)
  HP 4085B 48-pin or HP 4089A/B 96-pin Switching Matrix (SWM)

- DC Measurement Subsystem (DCS)

  HP 4142B Modular DC Source/Monitor

- Capacitance Measurement Subsystem (CMS)

  The CMS consists of one of the following instruments:

  HP 4280A 1 MHz C Meter/C-V Plotter (CMU)
  HP 4284A Precision LCR Meter (CMU84)

Figure 1-2 shows the basic configuration for the HP 4062UX equipped with a 48-pin SWM, including the system controller. Figure 1-3 shows the basic configuration for the HP 4062UX equipped with a 96-pin SWM, including the system controller.

Each subsystem and instrument is briefly described on the following pages. For more detailed information, see the *System Information* manual.

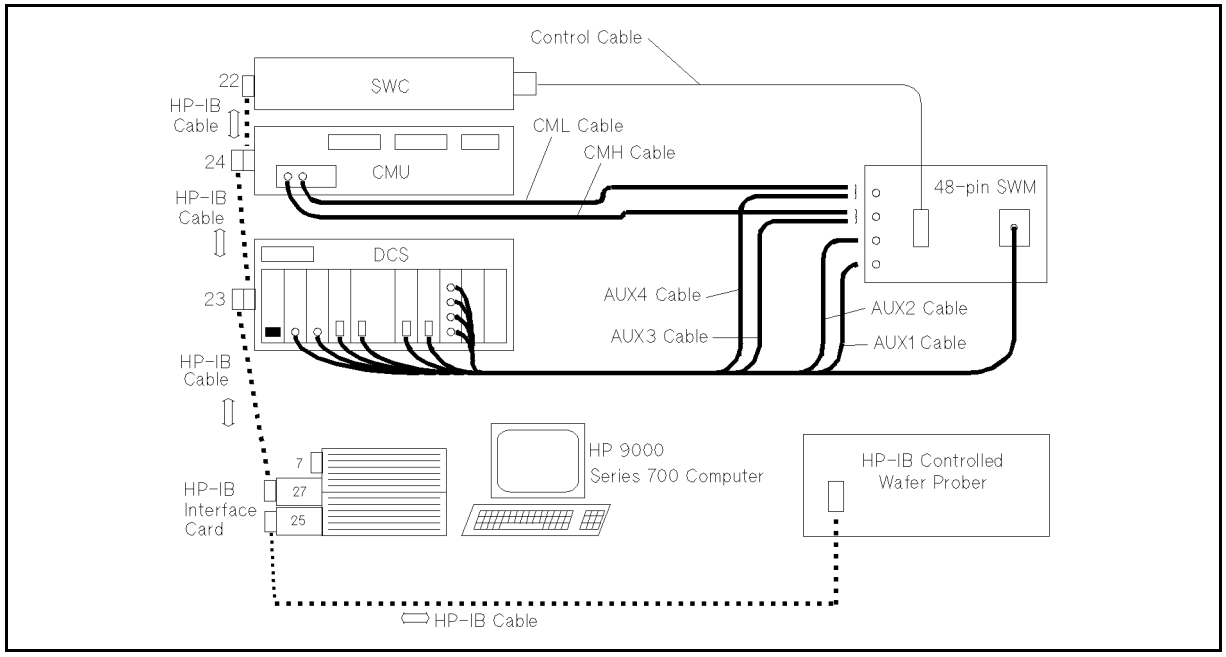


Figure 1-2. Standard System Configuration (48-pin SWM System)

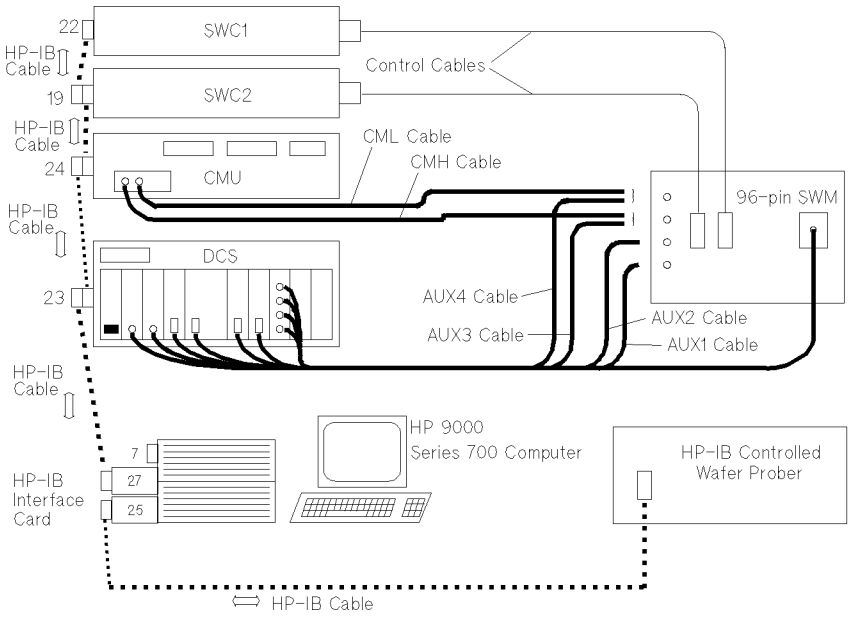


Figure 1-3. Standard System Configuration (96-pin SWM System)

Switching Matrix Subsystem (SMS)

Your SMS's configuration depends on your SWM (48- or 96-pin), as follows:

- If your SWM is a 48-pin SWM, your SMS includes the HP 4085B SWM and one HP 4084B SWC.
- If your SWM is the 96-pin SWM, your SMS includes the HP 4089A or HP 4089B SWM and two HP 4084B SWCs. One SWC controls measurement pins 1–48, and the other SWC controls measurement pins 49–96.

Switching Matrix Controller (SWC)

The SWC provides relay control signals and dc power for the SWM.

Switching Matrix (SWM)

The SWM determines the connection configuration between the measurement pins (measurement terminals) and the HP 4062UX's measurement subsystems and instruments.

You can install the SWM on a wafer prober, or you can connect any of the furnished test fixtures to the SWM. The SWM's switching relays enable programmable electronic switching to provide virtually any connection configuration between measurement subsystems and the SWM's measurement pins. When using a wafer prober, this capability lets you perform a variety of parametric measurements through the SWM with only one probing to a wafer.

To minimize leakage current, which affects low current measurements, the measurement signal lines inside the SWM are fully guarded. Also, to minimize the effects of residual resistance, you can implement a Kelvin connection at any SWM measurement pin.

All SWM switching relays are mounted on removable boards, called pin boards, to make it easy to replace a defective relay.

If your system has a port expander on the HP 4089B SWM (Option 340), you can use additional ports as auxiliary (AUX) ports. This option increases the number of AUX ports from 4 to 12. For more information on the port expander, see the *System Information*.

DC Subsystem (DCS)

The DCS handles dc measurement tasks through the HP-IB. The DCS's configuration depends on the combination of plug-in modules ordered.

Two types of DCS programmable Source/Monitor Unit (SMU) plug-in modules are available. The HP 41420A 200 V/1 A SMU and the HP 41421B 100 V/100 mA SMU make a wide range of high resolution I-V measurements possible. You can program each SMU to operate as a voltage source/current monitor (V source mode) or as a current source/voltage monitor (I source mode).

SMUs connected to SWM ports 2, 3, and 4 can be used to implement Kelvin connections, and all SMUs have a driven guard terminal. Because SMUs require no adjustments, are separate plug-in modules, and are identical—except for the HP 41420A 200 V/1 A SMU—the DCS is easy to maintain and repair.

Other available DCS plug-in modules are the HP 41424A Voltage Source/Voltage Monitor Unit (VS/VMU), which provides two voltage sources (VS1 and 2) and two voltage monitors (VM1 and 2), and the HP 41425A Analog Feedback Unit (AFU).

Warning

HP 41423A 1000 V High Voltage Unit (HVU) or HP 41422A 10 A High Current Unit (HCU) are plug-in modules of the HP 4142B DCS and can be installed in the DCS. However, the SWM is not guaranteed for voltage that exceeds ±200 V or current that exceeds ±1 A. If you force high voltage or current to the devices under test (DUTs) through the SWM, the SWM would not only be damaged but you might also get an electrical shock. Do not operate the HP 4062UX system if an HVU or HCU is installed in the DCS.

Capacitance Measurement Subsystem (CMS)

The CMU provides high speed capacitance and conductance measurements at 1 MHz. The CMU's internal dc bias source and timer let you perform accurate C-V, G-V, and C-t measurements.

The CMU84 provides capacitance and conductance measurements from 1 kHz to 1 MHz. The CMU84's internal dc bias source (Option 531) lets you perform accurate C-V and G-V measurements.

Your System Controller

The HP 4062UX's system controller is an HP 9000 Series 700 Computer. This manual assumes that you have HP 9000 Series 700 operating and programming experience.

If this is not the case, refer to the following manuals to thoroughly familiarize yourself with your system controller so you can make the most efficient use of this manual.

A Beginners Guide to HP-UX
Introducing UNIX System V

Software Concepts

The system software provides functions for C programming to control the HP 4062UX system instruments.

Test Instruction Set (TIS)

The Test Instruction Set (TIS) is a powerful set of library functions that facilitate measurement programming for the HP 4062UX. Measurement programs that include TIS are easier to maintain than programs written solely in the C language.

See Chapter 3 for details, and the *Programming Reference for C Library* for complete TIS synopsis and other rules governing the use of TIS.

Probing Control Library (PCL)

The Probing Control Library (PCL) is a set of functions used to control an automatic wafer prober. The PCL functions are used with the probing pattern data file created by the Probing Pattern Generator (PPG). PCL consists of functions that perform several tasks, such as getting a probing pattern data file, designating an initial probing position, specifying a chip and test module to be probed, and reporting the present probing position. PCL functions are used in the same way as the TIS functions and make it easy to control the wafer prober.

See Chapter 6 for details.

File Creation Library (FCL)

The File Creation Library (FCL) is a set of functions used to create data files for wafer maps and statistical graphics. Data files created by FCL are compatible with Basic Statics and Data Manipulation (BSDM).

See Chapter 6 for details.

Prober Drivers

The Prober Drivers are a basic set of functions used to control the wafer prober. There are several functions that can be used commonly among the supported wafer probers.

See Chapter 5 for details.

System Software Specifics for C Programming Library

- `/usr/pcs/bin/pcsstart` executive file

  This is a command to initialize the HP 4062UX system.

- `/usr/pcs/lib` directory

  The `libhp4062.a` is an archived file of the object code for the TIS, PCL, and FCL. The `libprober.a` is an archived file of the object code for the prober driver functions used commonly. The other files are archived files of the object code for the prober driver functions used specifically for a respective wafer prober model.

- `/usr/pcs/demo/c` directory

  This directory includes sample measurement programs written in C language.

- `/usr/pcs/src/prober` directory

  This directory includes source files of the prober drivers.

- `/usr/pcs/src/wafer` directory

  This directory includes source files of the PCL and FCL.

- `/usr/pcs/src/libs` directory

  This directory includes source files of the PGMLIB and other miscellaneous files.

- `/usr/include/pcs` directory

  This directory includes header files. The `tis.h` file includes the function declarations and macro definitions of the TIS. The `prober.h` file includes the function declarations and macro definitions of the prober drivers. The `wafer.h` file includes the function declarations and macro definitions of the PCL and FCL.

Parametric Test Session

Test Session Specifics

During normal system operation, several test sessions usually run concurrently, with multiple processes executing within each test session. Test sessions have their own attributes and all processes within each test session share these attributes independently from other test sessions. Each test session has the following attributes:

- Test Session Name

  Test session names distinguish each test session and are arbitrary strings limited to 14 characters.

- Online and Offline Mode

  Test session online or offline status applies to all processes within your test session.

- Instruments Settings

  Instrument settings are maintained and are referred to by TIS functions. For offline mode test sessions, TIS maintains and refers to instrument settings as if the instruments are actually present.

- System Configuration

  Your system configuration is registered by the startup command named `pcsstart`. Each offline test session can have its own system configuration.

- Locking Status (Explicit and Implicit)

  Refer to "Locking" for complete details on locking.

Figure 1-4 shows an example of typical multiple test session system operation, with multiple processes taking place within the test sessions.

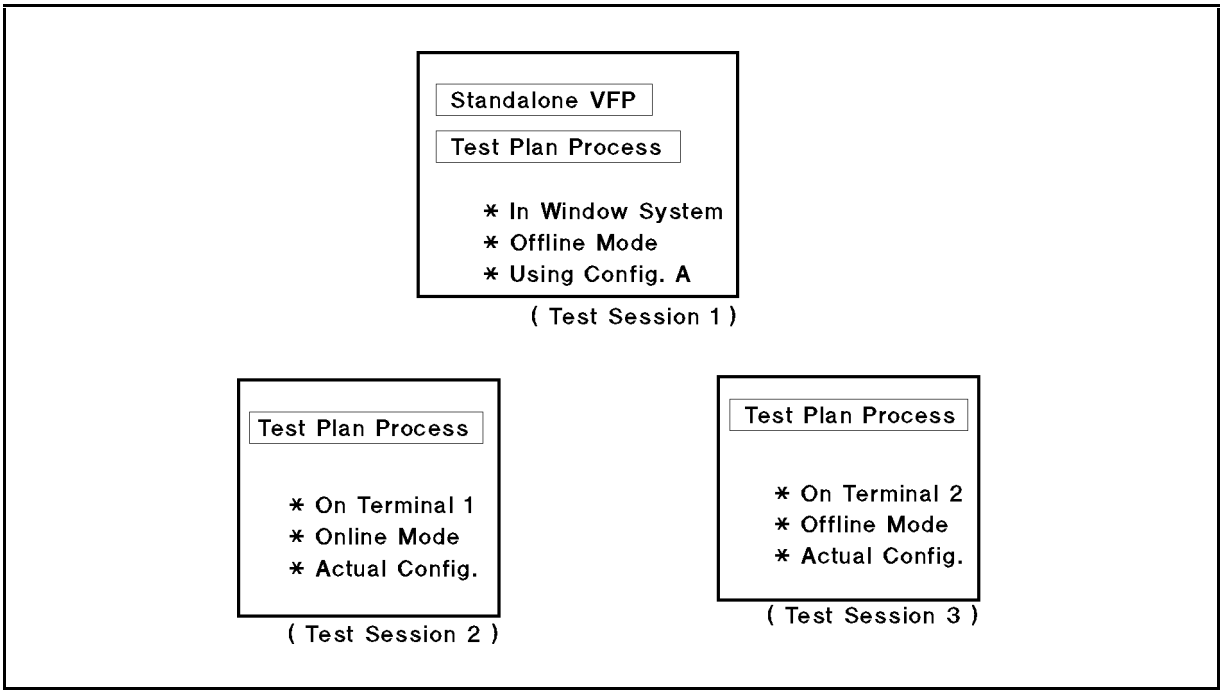


Figure 1-4. Typical Multiple Test Sessions and Processes System Operation

As Figure 1-4 shows, three independent test sessions are executing concurrently.

Test session 1 executes in the offline mode from the window environment using its own system configuration. In this test session, a measurement program is being debugged using the VFP (two processes). Note that the VFP is executed on the BASIC/UX.

Test session 2 executes in the online mode from terminal 1 using the actual system configuration.

Test session 3 executes in the offline mode from terminal 2, also using the actual system configuration.

Resource Management

The HP 4062UX software provides a resource management method for the C programming environment, called *locking*.

Locking

The locking function of the HP 4062UX "locks," or exclusively controls, and "unlocks," or releases control of, the system's resources—system instruments or wafer prober, or both—specified in a single TIS control function (*implicit* locking) or entire measurement program sequences (*explicit* locking).

Implicit and explicit locking are described in this section. Determine the locking method you will use by taking into consideration the tradeoffs between test execution speed and system flexibility. Figure 1-5 shows the basic concepts of implicit and explicit locking.

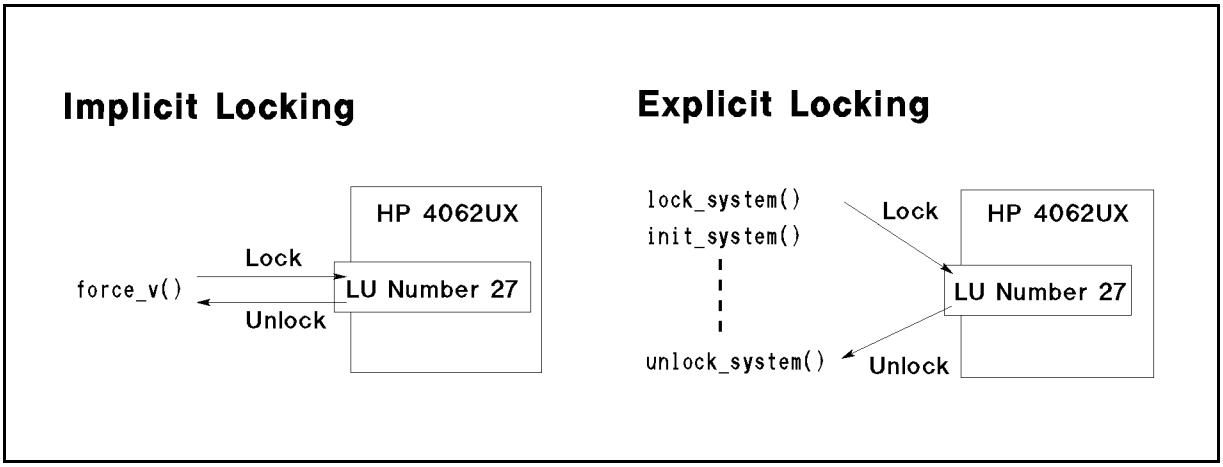


Figure 1-5. Implicit and Explicit Locking Concepts

- Implicit Locking

  Implicit locking locks the system resources to a TIS control function until the function executes, at which time the TIS control function unlocks the system resources. For example, in Figure 1-5, the system resources that are connected to an HP-IB interface with a logical unit number of 27 implicitly lock to the Force_v function when Force_v begins, then unlocks when Force_v executes.

  The system resources can implicitly lock to TIS control functions from other processes, but can implicitly lock to only one TIS control function at a time.

  If the system resources are implicitly locked to a TIS control function, succeeding TIS control functions will queue until the system resources are unlocked.

  If the system resources are explicitly locked (see below) to another process, however, TIS control functions that attempt to implicitly lock the system resources fail, and an error is reported.

- Explicit Locking

  Explicit locking exclusively locks and unlocks the system resources to and from a process via the Lock_system and Unlock_system TIS functions. For example, in Figure 1-5, the system resources that are connected to an HP-IB interface with a logical unit number of 27 explicitly lock to the process during the succeeding measurement program(s) at Lock_system, then unlock when the program sequence is completed and Unlock_system is executed.

  Explicitly locked system resources also unlock if the running process is over or aborted. If the system resources are explicitly locked to a process, TIS control functions that attempt implicit locking are ignored.

  Explicit locking helps to optimize program execution speed by eliminating frequent locking and unlocking operations.

- Locking the Wafer Prober

  Because your wafer prober is controlled by the Standard Instrument Control Library (SICL) routines, the wafer prober is locked and unlocked using I/O interface locking and unlocking functions: `ilock()` and `iunlock()`.

Turning the HP 4062UX System On and Off

This section covers the basic procedures for turning your HP 4062UX on and off.

Caution

Do *not* turn all the system instruments on or off at the same time. High power consumption may generate noise and affect other computers and instruments.

The power of the system controller, which is the workstation running the HP-UX operating system, usually remains on. When stopping the HP-UX and turning the system controller off, you must perform a shutdown procedure. For the shutdown procedure, see the *HP-UX System Administration Tasks* manual.

For a wafer prober, see the wafer prober manual.

Procedure to Turn the System On

When you turn the HP 4062UX system on, use the following procedure:

1. Open the front door of the system cabinet, and check that all instruments are turned off.
2. Turn on the SYSTEM ON/OFF switch, which is located at the lower right.
3. Turn each instrument on (see Figure 1-6). For the DCS, the LINE switch is located on the rear panel, and the POWER switch is located on the front panel.

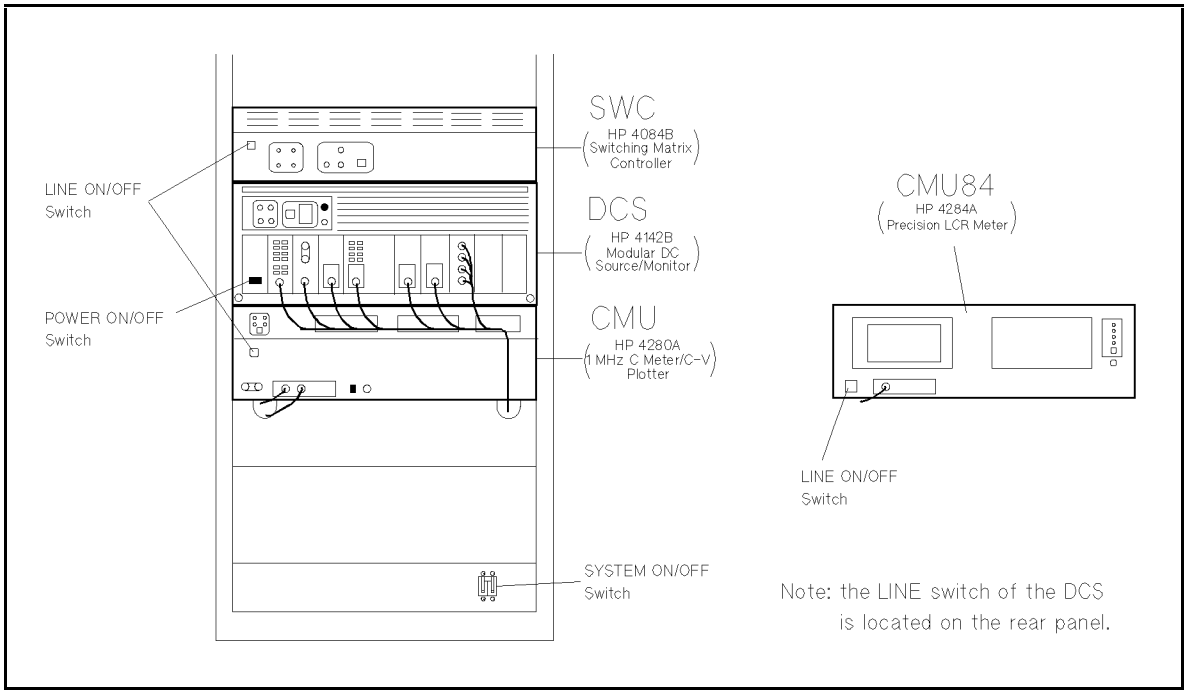


Figure 1-6. LINE Switch Location

After you turn the system instruments on, each instrument executes self-test. When an error occurs, turn the failed instrument off and turn it on again. If the error occurs again,

see the *Quick Maintenance Guide*, the HP 4280A *Operation and Service Manual*, the HP 4284A *Operation Manual*, and the HP 4142B *Operation Manual*.

4. Allow the system to warm up for a minimum of 40 minutes.
5. Run the system diagnostics program as described in the *System Information* manual.

Procedure to Turn the System Off

When you turn the HP 4062UX system off, use the following procedure:

1. Open the front door of the system cabinet.
2. Turn each instrument off (see Figure 1-6).
3. Turn off the SYSTEM ON/OFF switch.

HP-UX Environment Settings

To establish the test program development environment for the HP 4062UX system, the following two HP-UX environment variables should be properly set:

- LPATH variable includes the /usr/pcs/lib.
- PATH variable includes the /usr/pcs/bin.

The LPATH variable sets the library search path for the `cc` command (C compiler). If the LPATH variable is not set, the /lib and /usr/lib is used as the library search path.

The PATH variable sets the command search path.

Showing the LPATH and PATH Variables

To show what search paths are set in the LPATH and PATH variables, use the `env` command as follows:

```
$ env
  .   .   .   .   .   .   .   .
SHELL=/bin/ksh
LPATH=/usr/pcs/lib:/lib:/usr/lib                                    ⇐
  .   .   .   .   .   .   .   .
PATH=/bin:/usr/bin:/usr/contrib/bin:/usr/local/bin:/usr/pcs/bin:.   ⇐
TZ=PST8PDT
```

In the example above, the /usr/pcs/lib, /lib, and /usr/lib directory paths are set to the LPATH variable, and the /bin, /usr/bin, /usr/contrib/bin, /usr/local/bin, /usr/pcs/bin, and . (current directory) are set to the PATH variable.

Setting the LPATH and PATH

The LPATH and PATH variables are set in the /etc/profile and /etc/csh.login file when the HP 4062UX system is installed. The /etc/profile and /etc/csh.login file is the global profile file. The local profile file is .profile and .login in the home directory of each user account.

The directory paths for the LPATH and PATH variables are globally set. It is not necessary for each user to take care of these variables in the HP 4062UX environment.

2

Compilation and Execution

This chapter explains how to compile a test program in the C programming environment. In general, the `cc` command is used to compile programs written in C programming language. Before executing the test program, you must execute the `pcsstart` command to initialize the test session in the HP 4062UX test environment.

The following topics are included in this chapter:

- How to compile the test program.
- How to execute the `pcsstart` command and the test program.
- The `pcsstart` command.

Figure 2-1 shows what this chapter covers in the flow chart for developing test programs. The topics in the shaded squares are explained in this chapter. You can skip "Executing the test program in offline mode" if no other user uses the HP 4062UX system.

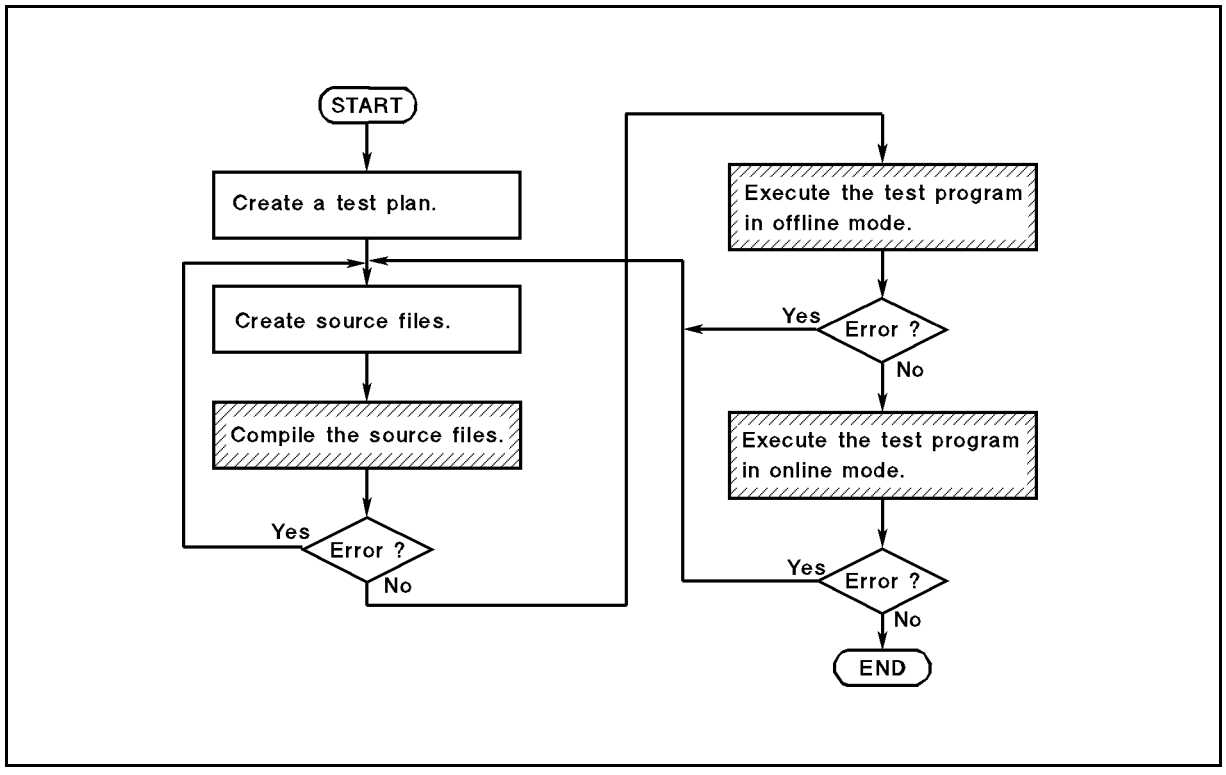


Figure 2-1. Flow Chart for Developing Test Program: Compiling & Executing

Compiling a Test Program

In general, program source files written in the C programming language are compiled using the `cc` command. The `cc` command can compile a test program written in the C programming language for the HP 4062UX system. The following explains how to compile test programs.

CC Command

The `cc` command in the HP-UX system compiles the test programs and links the necessary libraries. The following shows the command line syntax for the HP 4062UX system:

```
$ cc prog.c [mylib.o ... ][-g][-o outfile]
[-lprober [-legx][-legy][-ltsk][-ldmyprob]]-lhp4062 -lsicl [-lm]
-lcl
```

The command line options are:

prog.c	C source file of the test program to be compiled.
mylib.o	Object files, which are generated by the `cc` command with `-c` option, to be linked. The `-c` option forces the C compiler to compile only.
-g	Causes the compiler to generate additional information needed by the symbolic debugger.
-o *outfile*	Names the output file from the linker. This file is executable. The default name is `a.out`.
-lprober	Links the prober driver general library.
-legx	Links the EGX prober driver library.
-legy	Links the EGY prober driver library.
-ltsk	Links the TSK prober driver library.
-ldmyprob	Links the DUMMY prober driver library.
-lhp4062	Links the HP 4062 library (TIS, PCL, FCL, and PGMLIB).
-lsicl	Links the SICL.
-lm	Links the mathematics library. This library must be linked when the PCL, FCL, or prober driver library is need.
-lcl	Links the Pascal runtime library.

To use the `-l`*xxxx* option in the command line, the LPATH environment variable must be set properly. See "HP-UX Environment Settings" in Chapter 1 for how to set the LPATH environment variable.

You can use the library file name with an absolute path name instead of the `-l`*xxxx* options. The library file name and its absolute path name are as follows:

`-lprober`	`/usr/pcs/lib/libprober.a`
`-legx`	`/usr/pcs/lib/libegx.a`
`-legy`	`/usr/pcs/lib/libegy.a`

```
-ltsk      /usr/pcs/lib/libtsk.a
-ldmyprob  /usr/pcs/lib/libdmyprob.a
-lhp4062   /usr/pcs/lib/libhp4062.a
-lsicl     /usr/lib/libsicl.a
-lm        /lib/libm.a
-lcl       /usr/lib/libcl.a
```

For example, you can enter the following command line to compile:

```
$ cc test1.c /usr/pcs/lib/libhp4062.a /usr/lib/libsicl.a /usr/lib/libcl.a
```

is equivalent to

```
$ cc test1.c -lhp4062 -lsicl -lcl
```

Compiling Example

The example program is stored in the file "`spot.c`" under the `/usr/pcs/demo/c` directory. The `spot.c` is a test program to measure a 1 kΩ resistance in the system test module.

The following procedure is used to compile the `spot.c` file and to create the executable file `spot`:

1. Create a directory named "`4062demo`" in your home directory as follows:

   ```
   $ mkdir 4062demo
   ```

2. Move the current directory to the `4062demo`.

   ```
   $ cd 4062demo
   ```

3. Copy the file `spot.c` from the `/usr/pcs/demo/c` directory.

   ```
   $ cp /usr/pcs/demo/c/spot.c .
   ```

4. Compile `spot.c` and create the executive file `spot`.

   ```
   $ cc spot.c -o spot -lhp4062 -lsicl -lcl
   ```

If no error is reported, the compilation finished successfully.

Executing the PCSSTART Command and Test Program

To execute the test program, a parametric test session must be established in the login environment. The parametric test session enables the test program to control the system instruments.

For example, the following procedures are the compilation and execution for test program "`spot.c`," which is stored in the `/usr/pcs/demo/c` directory. The "`spot.c`" program measures 1 kΩ resistance in the system test module.

1. Login as user "hp4062" (example):

   ```
   Login: hp4062
   ```

2. Create a directory named "`4062demo`" (example) to store the `spot.c` file:

   ```
   $ mkdir 4062demo
   ```

3. Copy the `spot.c` source file to the `4062demo` directory:

   ```
   $ cp /usr/pcs/demo/c/spot.c 4062demo
   ```

4. Change the current directory to `4062demo`:

   ```
   $ cd 4062demo
   ```

5. If the HP 4062 system is equipped with a 96-pin SWM, you must edit `spot.c` using an editor, such as the `vi` editor, and change the pin assignments. For details, see "Spot I-V Measurements" in Chapter 3.

6. Compile test program "`spot.c`" and create an output file "`spot`," which is an executive file. Confirm that "`spot`" compiled successfully:

   ```
   $ cc spot.c -o spot -lhp4062 -lsicl -lcl
   ```

7. Install the test fixture only to the SWM.

8. Execute the `pcsstart` command to initialize the HP 4062UX system:

   ```
   $ pcsstart -on
   ```

   The `-on` option sets the execution mode to online.

9. For the 48-pin SWM system, remove the test fixture adapter and install the system test module. For the 96-pin SWM system, remove the test fixture adapter, install the system test module to the test fixture adapter, and install it on the SWM.

10. Execute the test program "`spot`:"

    ```
    $ spot
    current =  0.00050 ampere (at  0.50000 volt)
    resistance = 1003.456 ohm
    ```

If the test program is executed successfully, about 1 kΩ is displayed.

The PCSSTART Command

The `pcsstart` command:

- Generates or initializes the parametric test session.
- Determines the execution mode (online or offline).
- Verifies the system configuration, if online mode.
- Initializes the system instruments, if online mode.

When you execute the `pcsstart` command, the codewords in the CODEWORD file are compared with the codeword generated by the ID module. If the codewords in the CODEWORD file are different, `pcsstart` reports an error and aborts.

Before you execute the `pcsstart` command, the prestart conditions must be satisfied. For the prestart conditions, see the "The START Program" in the Chapter 1 of the HP 4062C/UX *System Operation* manual.

When to Execute PCSSTART

You must execute `pcsstart` to use the HP 4062UX system once after you log in. You must also execute `pcsstart` after any of the following actions:

- When you turn the HP 4062UX system on.
- If you turn a system instrument off or on after you execute `pcsstart`.
- If you rearrange system cabling after you execute `pcsstart`.
- If you change the plug-in module configuration in the DCS after you execute `pcsstart`.
- If you change the pin board configuration in the SWM after you execute `pcsstart`.
- If you want to change the execution mode (online or offline).

PCSSTART Command Line Syntax

The `pcsstart` command has the following command line:

```
$ pcsstart [-q][-v]{ [-on]
                   { [-off [file]] }
```

Options:

-q	Quick operation. The DCS calibration is omitted.
-v	Verbose mode. Reports the system configuration information to the standard output.
-off [*file*]	Switches to offline mode. This option must not be used with -on option. The optional *file* gives a file name of dummy configuration. If the *file* is omitted, the file `/usr/pcs/etc/DUMCONFIG` is used.
-on	Switches to online mode. This option must not be used with the -off option.

Executing PCSSTART

Reporting the System Configuration

The `pcsstart` command can report the system configuration of the HP 4062UX system. To do so, use the following command line:

```
$ pcsstart -on -v -q
```

The -on option switches the execution mode to the online mode, -v sets the verbose mode, and -q omits the DCS calibration. Figure 2-2 shows an example report of the system configuration by the `pcsstart` command.

```
$ pcsstart -on -v -q
***************** SYSTEM CONFIGURATION *****************
4084B/4085B SWITCHING MATRIX:               Present
4142B DC SOURCE MONITOR (Rev. 4.00):        Present
4280A 1MHz C METER/C-V PLOTTER:             Not present
4284A 20Hz-1MHz LCR METER (Opt. 001/006):   Present
****************** DCS CONFIGURATION *******************
SLOT1:41421B(SMU1)          SLOT5:41421B(SMU4)
SLOT2:                      SLOT6:41424A(AUX1/2)
SLOT3:41420A(SMU2)          SLOT7:
SLOT4:41421B(SMU3)          SLOT8:41425A
*********** AUXILIARY PORT CONFIGURATION ***************
AUX1: VS1                   AUX3: CMH84
AUX2: VS2                   AUX4: CML84
******************** AVAILABLE PINS ********************
01 02 03 04    05 06 07 08    09 10 11 12    13 14 15 16
17 18 19 20    21 22 23 24    25 26 27 28    29 30 31 32
33 34 35 36    37 38 39 40    41 42 43 44    45 46 47 48
$
```

Figure 2-2. An Example: System Configuration Report

The following explains each section of the system configuration report.

- SYSTEM CONFIGURATION

  This section shows which system components are correctly set (`Present`) for system operation. If an instrument is displayed as `Not present`, make sure the instrument is turned on, check the instrument's HP-IB address setting and HP-IB cable connections, and execute the `pcsstart` again. The system controller conducts a serial poll of the individual system components on the HP-IB during `pcsstart`.

  If a port expander (Option 340) is equipped with an SWM, the "`(Opt. 340)`" is shown after the `SWITCHING MATRIX`.

  The `4142B DC SOURCE MONITOR` description shows the firmware ROM revision number. If the number is under 3.00, you can force up to 20 V to the search port when you perform analog search measurements.

The `4284A 20Hz-1MHz LCR METER` description shows Option 001 Power Amplifier/DC Bias and Option 006 2 m/4 m Cable Length Operation if the option is present. If Option 001 is not present, you can not perform C-V measurements. If Option 006 is not present, the measurement accuracy is not specified.

- DCS CONFIGURATION

  This section shows each DCS slot number, the model number of the plug-in module installed in each slot, and the SWM port (in parentheses) to which each module is connected. If an SWM port is not displayed, that module is not connected to the SWM. Note that the HP 41420A SMU occupies 2 DCS slots and `pcsstart` "sees" only the higher of the 2 slots where the SMU is installed (in this example, slot 3). Vacant slots are left blank (slot 7). If you reconfigure any DCS plug-in module after you execute `pcsstart`, you *must* execute `pcsstart` again.

- AUXILIARY PORT CONFIGURATION

  This section shows the configuration of the SWM's AUX ports. If the AUX ports are connected incorrectly, `Nothing Connected` is displayed. If you reconfigure any AUX port cables after you execute `pcsstart`, you *must* execute `pcsstart` again.

  If your SWM is an HP 4089B with Option 340 (adds a port expander), the AUXILIARY PORT CONFIGURATION block may vary. For instance, it may be displayed as follows:

```
  .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
********** AUXILIARY PORT CONFIGURATION ***************
AUX11:VS1  12:     13:VM1   AUX31:CMH84 32:     33:
AUX21:VS2  22:     23:VM2   AUX41:CML84 42:     43:
******************** AVAILABLE PINS ********************
  .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
```

  A blank field means either no instrument or an unsupported instrument is connected.

- AVAILABLE PINS

  This section lists the available measurement pins (pin boards) installed in the SWM. Pin boards not present at `pcsstart` execution are not listed (are not displayed).

Switching to Online mode

To move the execution mode to the online mode, the following command line is used:

```
$ pcsstart -on
```

If the -v option (verbose mode) is added in options, the system configuration information is also displayed. You can check the system configuration.

Switching to Offline mode

To move the execution mode to offline mode, the following command line is used:

```
$ pcsstart -off
```

If the -v option (verbose mode) is added in the option line, the system configuration information used in offline mode is also displayed. You can check the system configuration.

To specify the system configuration file for offline mode, add a file name after the -off option, for example, `-off DUMFILE`.

For the system configuration file for offline mode, see "Changing the System Configuration in Offline Mode" in Chapter 4. For information on how to execute the test program in offline mode, see "Offline Debugging" in Chapter 4.

PCSSTART and START

The `pcsstart` command is used in the HP-UX command interpreter environment to initialize the test session of the HP 4062UX system. On the other hand, the START program is used in the BASIC/UX environment for the same purpose. Some differences exists between the `pcsstart` command and START program.

The functions of the `pcsstart` command are shown in "The PCSSTART Command". The START program executes not only the `pcsstart` command functions but the following functions:

- Initializes a wafer prober connected to the HP 4062UX system.
- Initializes the COM blocks corresponding to the PCL, FCL, and prober drivers.

You must use Prober_init function in the prober driver to initialize a wafer prober. For the Prober_init function, see Chapter 5.

In the X11 window environment, the START execution in the BASIC/UX window can be used to initialize the C programming environment for the HP 4062UX instead of the `pcsstart` execution. Figure 2-3 shows a X11 window screen to illustrate the START execution. Once START is executed in the BASIC/UX window, you do not need to execute the `pcsstart` command.

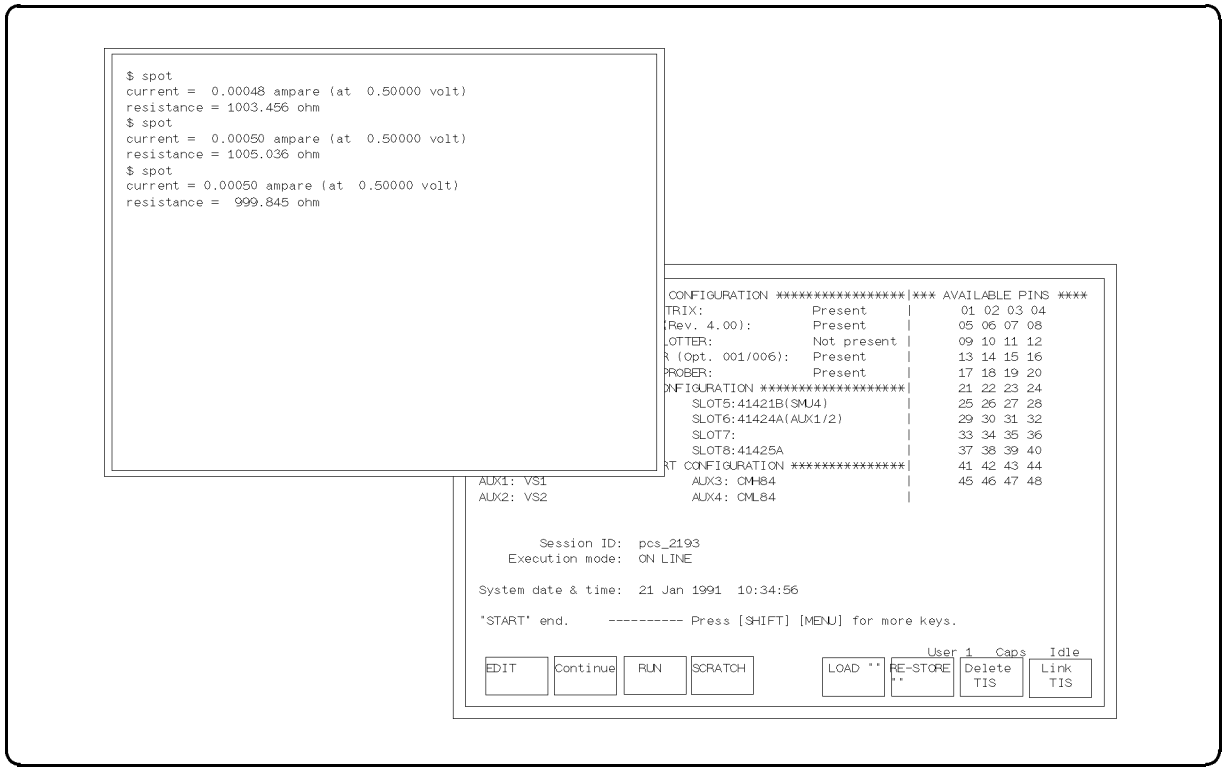


Figure 2-3. Executing the START Program in BASIC/UX Window

3

Measurement Programming

Introduction

The HP 4062UX system supplies the C library to help you develop test programs. This chapter covers how to program for various measurement methods using the TIS functions.

The library codes of TIS functions are archived in the `/usr/pcs/lib/libhp4062.a` file. The header file `tis.h` in the `/usr/include/pcs` directory must be included at the beginning of your test program to use the TIS functions. The SICL codes are archived in the `/usr/lib/libsicl.a` file. The header file `sicl.h` in the `/usr/lib` directory must be included at the beginning of your test program to use the SICL. Include the header files as follows:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <sicl.h>
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
```

Before introducing the measurement method, the typical program flow for a measurement is shown as follows:

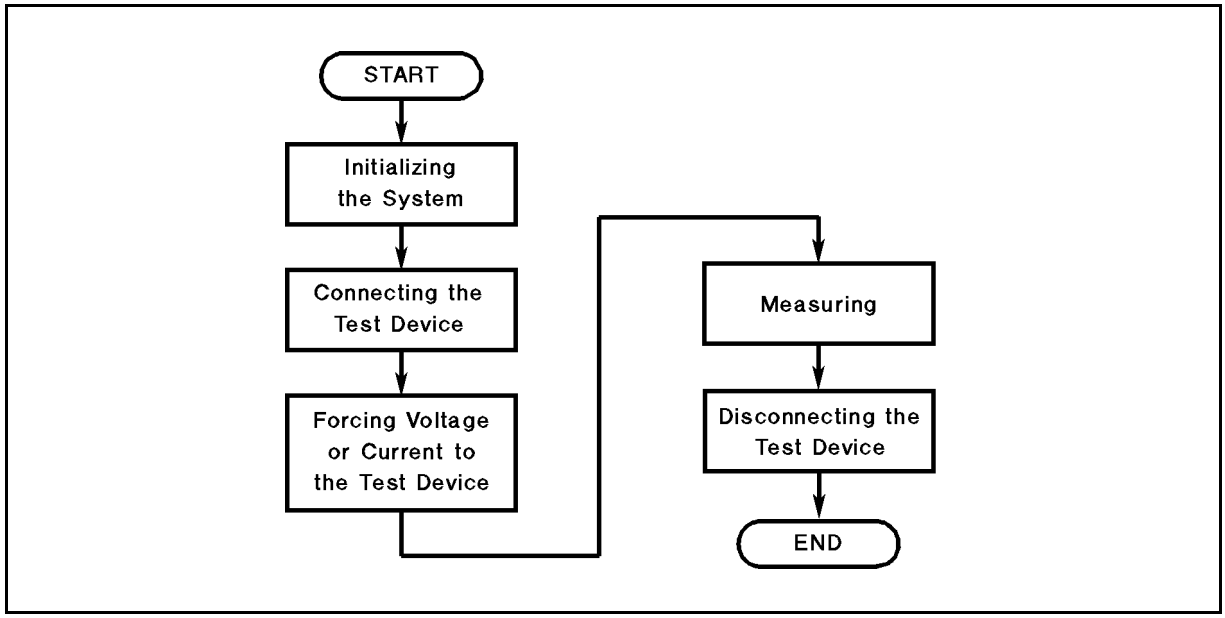


Figure 3-1. Typical Measurement Program Flow

Of course, the `pcsstart` command must be executed before you execute your measurement program. See "The PCSSTART Command" in Chapter 2 for details.

Before Controlling System Instruments

To bring the system instruments under the control of the system controller in which the HP-UX is installed, the following procedures are needed in the program:

- Constructing the data path to communicate with the system instruments.
- Locking the system instruments to the invoked process, if needed.
- Initializing the system instruments and TIS library.

In this section, the procedures and a brief explanation about the above items are provided. The following list provides the function names introduced in this section and brief descriptions of each:

Open_tis_hpib_fd	Searches for the proper logical unit number for the HP-IB interface in the `/usr/pcs/etc/SYSCONFIG` file, and the logical unit number is passed to the Set_tis_hpib_fd function.
Set_tis_hpib_fd	Makes the TIS functions use the HP-IB interface connected to the system instruments using the corresponding session ID gotten by the Open_tis_hpib_fd function.
Lock_system and Unlock_system	Control the interface access not to conflict the access for the system instruments.
Init_system	Initializes the system instruments and TIS library to control the system instruments.

Preparing to Access the System Instruments

To control the system instruments, the logical unit number in the /usr/pcs/etc/SYSCONFIG file must be gotten by Open_tis_hpib_fd function. Following is an example for connecting with the system instruments.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <sicl.h>

main()
{
    INST fd;
    . . . . . .
    . . . . . .
    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);
    . . . . . .
    . . . . . .
}
```

The `error_rep()` processes the errors when errors are detected in the `open_tis_hpib_fd()` routine.

For more details, refer to "Using HP SICL" in the *HP Standard Instrument Control Library User's Guide*.

Locking System Instruments to the Process

When more than one program or user tries to access system instruments simultaneously, conflicts arise and important control commands or measured data can be lost.

This problem is solved by locking the system instruments to the invoked process. The Lock_system function locks the HP-IB interface occupied by the system instruments so that other processes cannot access the system instruments. Unlock_system function releases the locking.

The Lock_system function has a parameter: the *wait* flag. If *wait* is set to 1, the Lock_system function waits until the locked process releases the system instruments. If the *wait* is set to 0 and if the system instruments are already locked by another process, Lock_system function sets the error code to the TIS variable `tis_errno` and returns −1.

For example, the Lock_system function is used as follows:

```
#include <stdio.h>
#include <pcs/tis.h>

main()
{
    . . . . . .
    . . . . . .
    if (lock_system(0) == -1) error_rep();
    . . . . . .
    . . . . . .
}
```

When the program is finished, the locking is released automatically.

Initializing System Instruments

The Init_system function is used to initialize the system instruments. The Init_system function clears all previous settings, except the setting for the PCL, FCL, and prober drivers, and the system conditions are set as follows:

Table 3-1. Initial Settings of HP 4062UX

Subsystem	Item	Settings
SWM	Measurement pins	Disconnected
DCS	SMU	VS mode, 0 V output, 20 V range, 100 μA current compliance, Filter on
	VM	20 V range, Grounded measurement mode
	VS	0 V output, 20 V range
	Integration time	SHORT
	Auto calibration	ON
	Output switch	OFF
CMU	C-G range	Auto range
	Test signal level	30 mVrms
	Integration time	MEDIUM
	DC bias source	0 V output, Internal dc bias, V output lamp on (enable)
CMU84	Measurement range	Auto range
	Test signal level	30 mVrms
	Integration time	MEDIUM
	DC bias source	0 V output
	Frequency	1 MHz

For example, Init_system function is performed after Set_tis_hpib_fd function, as shown below:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <sicl.h>

main()
{
    INST fd;
    . . . . . .
    . . . . . .
    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);
    if (init_system() == -1) error_rep();
    . . . . . .
    . . . . . .
}
```

For more details, see the "Technical Information" of the *System Information*.

Controlling SWM

After initializing the system instruments, all the connections between ports and pins are disconnected. The test program must make the necessary connections between ports and pins before forcing voltage or current. To control the connections between a port and pins, the following functions can be used:

- Connect_pin
- Connect_pin2
- Connect_pin3
- Connect_th
- Disconnect_all

For details, see the *Programming Reference for C Library*.

Connecting Ports and Pins

To make a connection between a port and a measurement pin, the Connect_pin function is used. For example, the following program lines make a connection between the SMU1 port and pin 32:

```
   .  .  .  .  .  .  .
int err;
   .  .  .  .  .  .  .  .  .  .  .
err = connect_pin(fnsmu(1), 32);
   .  .  .  .  .  .  .  .  .  .  .
```

If the `connect_pin()` function successfully makes the connection, `connect_pin()` returns 0 to the variable `err`. If not successful, −1 is returned to the variable `err`. For more about error handling, see "Error Handling" in Chapter 4.

The `fnsmu()` function returns a port address for the specified SMU port. You can use `32701`, which is the port address of the SMU1 port, instead of `fnsmu(1)` to specify the SMU port in `connect_pin()`. However, it's not easy to associate the port address `32701` with the SMU1 port number.

Table 3-2 lists the port address functions that convert the number (port number, SMU number, AUX number, and so forth) to a port address.

Table 3-2. Port Address Functions

Function	Port Name	Parameter Meaning	Range of Parameter
`fnport(` *number* `)`	all port	port number	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, −1, −8, and −9 21–32[1]
`fnsmu(` *number* `)`	SMU port	SMU number	1–4, 29[1], and 32[1]
`fnaux(` *number* `)`	AUX port	AUX number	1–4, 11–13[1], 21–23[1], 31–33[1], and 41–43[1]
`fngnd()`	GNDU port	—	—
`fncmh()`	CMH port	—	—
`fncml()`	CML port	—	—
`fngsmu()`	GSMU port	—	—
`fngcmu()`	GCMU port	—	—

1 These numbers are available when the SWM has a port expander.

For the port addresses, see the Fnport function reference page of the *Programming Reference for C Library*.

To make connections between a port and two measurement pins, use the Connect_pin2 function. To make connections between a port and three measurement pins, use the Connect_pin3 function. Use these functions as follows:

```
  .  .  .  .  .  .  .  .
int err;
  .  .  .  .  .  .  .  .
err = connect_pin2(fnsmu(1), 12, 16);
err = connect_pin3(fngnd(), 28, 32, 36);
  .  .  .  .  .  .  .  .
```

To make connections between a port and a series of pins, use the Connect_th function, as follows:

```
  .  .  .  .  .  .  .  .
int err;
  .  .  .  .  .  .  .  .
err = connect_th(fnsmu(3), 37, 48);
  .  .  .  .  .  .  .  .
```

Disconnecting the Connections

To disconnect the connection between a port and a measurement pin, the Connect_pin function is also used. When 0 is specified as the parameters of Connect_pin function, the function disconnects the specified connections. Examples are as follows:

```
. . . . . .
int err;
. . . . . .
err = connect_pin(0, 16);          disconnecting pin 16
err = connect_pin(fnsmu(4), 0);    disconnecting all pins connected to SMU4 port
err = connect_pin(0, 0);           disconnecting all connections
. . . . . .
```

To disconnect all the connections, use the Disconnect_all function, which is the same as `connect_pin(0,0)`. An example follows:

```
. . . . . .
int err;
. . . . . .
err = disconnect_all();
. . . . . .
```

General I-V Measurements

In this section, the following topics are covered:

- To force or measure a specified value, use the Force_i, Force_v, Measure_i, Measure_v, or Force_meas function.
- To set the parameters of a pulse measurement, use the Set_pbias function, and to trigger and measure the value, use the Measure_p function.
- To set the integration time and filter mode of the SMUs and VMs, use the Set_smu function.
- To set the measurement mode of the VMs, use the Set_vm function.

Spot I-V Measurements

The spot measurement is one of the most basic measurements: forcing a voltage or current and measuring the current or voltage at a unit of the DCS. The Force_v or Force_i function forces the specified voltage or current, respectively. The Measure_v or Measure_i function measures the voltage or current at the specified port. After the measurement, Disable_port function is used to clear the output of the SMUs or VSs.

There is an example to measure a resistance (1 kΩ) in the HP 16076A or HP 16356A System Test Module. Figure 3-2 shows the measurement circuit and Figure 3-3 lists the test program named "`spot.c`," which is stored in the `/usr/pcs/demo/c` directory. Note that the "`spot.c`" program is for 48-pin SWM systems, and pin numbers 20 and 24 must be used for the 96-pin SWM systems, instead of pin numbers 12 and 16 . To compile and execute the "`spot.c`" program, see the Chapter 2.

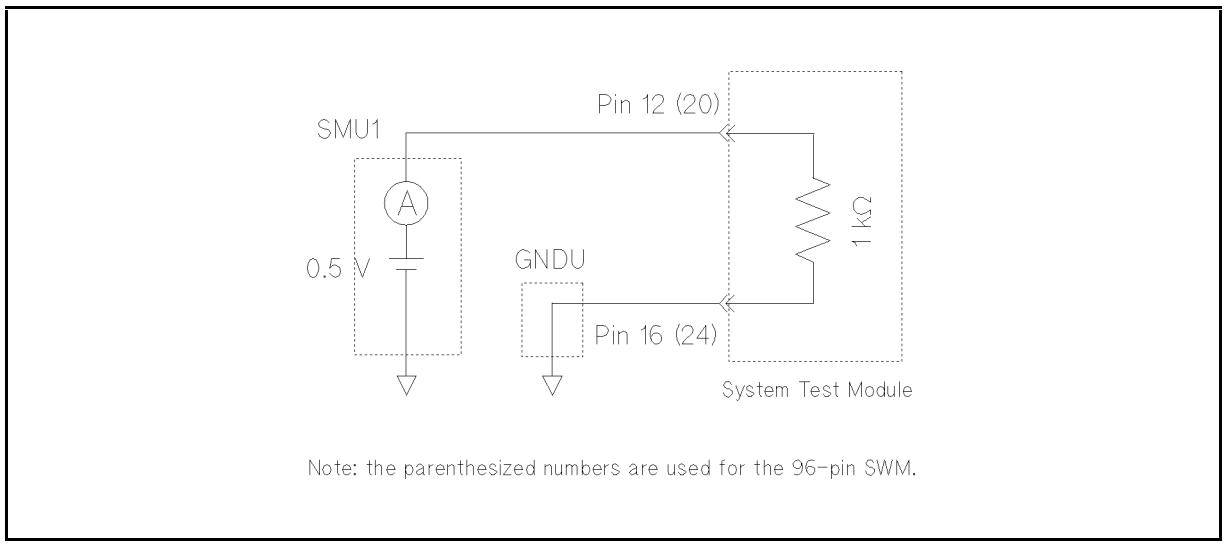


Figure 3-2. Schematic for Measurement Circuit: "spot.c"

```
 1 #include <stdio.h>
 2 #include <pcs/tis.h>
 3
 4 #ifdef __hp9000s700
 5 #include <sicl.h>
 6 #endif
 7
 8 main()
 9 {
10 #ifdef __hp9000s700
11      INST fd;
12 #else
13      int fd;
14 #endif
15
16     double current;
17
18     if ((fd = open_tis_hpib_fd()) == -1) error_rep();
19     set_tis_hpib_fd(fd);
20
21     if (init_system() == -1) error_rep();
22     if (lock_system(0) == -1) error_rep();
23
24     if (connect_pin(fnsmu(1), 12) == -1) error_rep();
25     if (connect_pin(fngnd(), 16) == -1) error_rep();
26
27     if (force_v(12, 0.5, 2.0, 1E-3) == -1) error_rep();
28     if (measure_i(12, &current, 1E-3) == -1) error_rep();
29
30     if (disable_port(12) == -1) error_rep();
31     if (connect_pin2(0, 12, 16) == -1) error_rep();
32
33     printf("current = %8.5f ampare (at %8.5f volt)\n", current, 0.5);
34     printf("resistance = %8.3f ohm\n", 0.5 / current);
35 }
36
37 int error_rep()
38 {
39     int errn[2];
40     char errm[100];
41
42     error_info(errn, errm);
43     fprintf(stderr, "%s\n", errm);
44
45     exit(-1);
46 }
47
```

Figure 3-3. Sample Program for Spot Measurement: “spot.c” (48-pin SWM System)

The following are explanations for the above example:

Line 1 to 6	Declares the include files, which are called header files.
Line 18 and 19	Open_tis_hpib_fd searches the SYSCONFIG file to get the session ID, which is set to allow communication with system instruments.
Line 21 and 22	Initializes the system instruments and lock the system instruments to this process.
Line 24 and 25	Connects the SMU1 port to measurement pin 12 and connect the GNDU port to the measurement pin 16 to connect each unit to the terminals of the resistor.
Line 27	Forces 0.5 V at 2.0 V range with 1 mA compliance.
Line 28	Measures the current value at 1 mA range.
Line 30	Forces 0 V from SMU1, that is, clears SMU1.
Line 31	Disconnects the connections of measurement pins 12 and 16.
Line 33 and 34	Prints the measured current and calculated resistance on the screen.
Line 37 to 46	Are a subroutine to deal with run-time errors. This subroutine prints the error message on the screen. For more about "`stderr`," see a text book on the C programming language.

The Force_meas function forces a specified voltage/current to a specified port or measurement pin, and measures the voltage/current value at the specified port or measurement pin. If the source port is not the same as the measurement port, the source mode of the measurement port must be previously set using Force_v or Force_i function.

Pulsed I-V Measurements

To force a pulsed wave to a device under test (DUT) and to measure a value synchronized with the pulsed wave, the Set_pbias and Measure_p functions are used. The Set_pbias function sets the pulse base, pulse peak, pulse width, pulse period, hold time, and compliance to the specified port or pin. The Measure_p function triggers the output of the pulse wave and measures the value at a specified port or pin.

The schematic of the wave form for a pulsed measurement is shown in Figure 3-4.

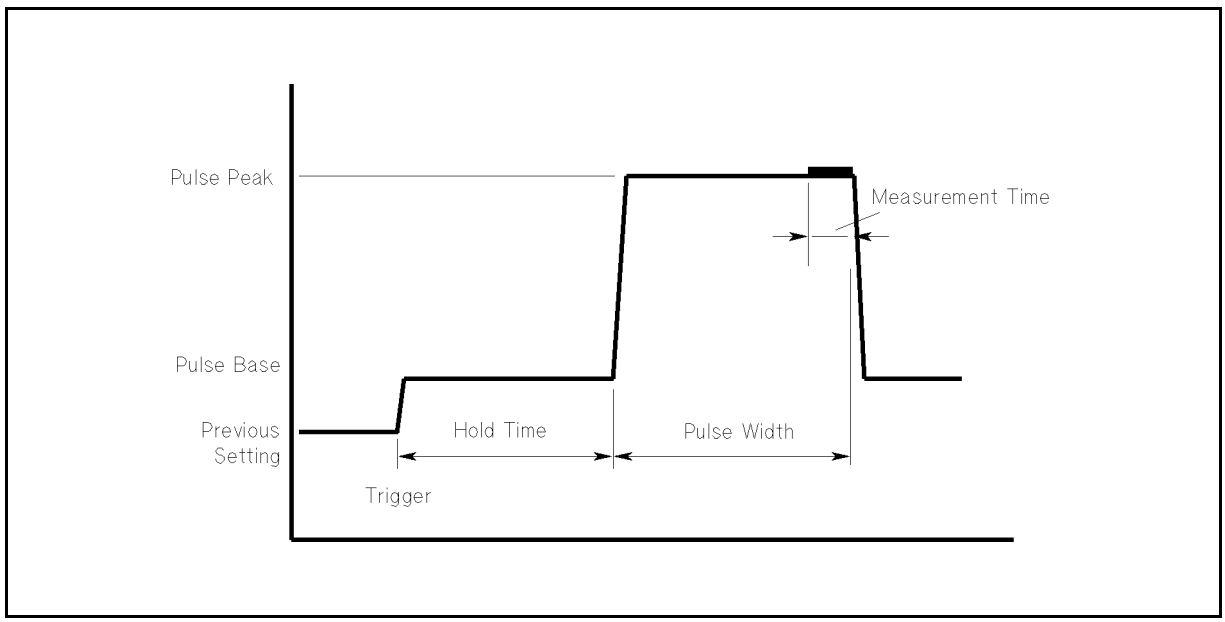


Figure 3-4. Wave Form of Pulsed I-V Measurements

For example, the following program lines show the pulse output and measurement section of the program.

```
  .  .  .  .  .  .
int err;
  .  .  .  .  .  .
err = set_pbias(12, V_SOURCE, 20., -3., 15., 0.4, 1., 0.5, 1E-3);
err = measure_p(12, I_MEAS, &current, 1E-3);
  .  .  .  .  .  .
  .  .  .  .  .  .
```

Setting SMUs, VSs, and VMs

The Set_smu function is used to set the integration time of the SMUs and VMs and to set the filter mode of the SMUs in the DCS. The Set_vm function is used to set the measurement modes of the VMs.

The integration time can not be set for each SMU or VM separately, but must be set for all the SMUs and VMs at one time. The integration time can be set to SHORT, MEDIUM, LONG, or MANUAL. When the integration time is set to MANUAL, anything from 1 time to 1023 times can be set.

The filter is used to reduce overshoot voltage or current in the output of the SMU. To prevent damage to the DUTs from overshoot voltage or current, the filter mode must be set to on. A filter is not inherent in VS.

The measurement mode of the VMs can be set to grounded measurement or differential measurement mode. For more details on the measurement mode of VMs, see "Theory of Operation" of the *System Information*.

For example, to set the integration time to LONG and to set the filter mode to on, the following program lines are available:

```
  .   .   .   .   .   .
int err;
  .   .   .   .   .   .
err = set_smu(INTEG_LONG, FILTER_ON, 0);
  .   .   .   .   .   .
  .   .   .   .   .   .
```

To set the measurement mode of the VMs to the differential measurement mode, the following program lines are available:

```
  .   .   .   .   .   .
int err;
  .   .   .   .   .   .
err = set_vm(fnaux(1), VM_DIFF);
  .   .   .   .   .   .
  .   .   .   .   .   .
```

General C·G Measurement (CMU)

This section covers how to measure capacitance and conductance of the DUTs by using a CMU (HP 4280A 1 MHz C Meter/C-V Plotter). The topics of this section are as follows:

- To measure the capacitance and conductance, use the Measure_cmu function.
- To force a bias voltage, use the Force_v function.
- To set integration time and test signal level for the CMU, use the Set_cmu function.

Note

Before measuring the capacitance of the DUTs, stray capacitance from the cables and SWM must be measured. After measuring the capacitance of the DUTs, you must subtract the stray capacitance from the measured capacitance. For details, see "Technical Information" in *System Information*.

C·G Measurements

To measure the capacitance and conductance using a CMU, use the Measure_cmu function. After the measurement, the Disable_port function must be used to clear the CMU.

Figure 3-5 and Figure 3-6 show an example of measuring capacitance (100 pF) in the HP 16076A or HP 16356A System Test Module. The program named "`cg.c`" is stored in the `/usr/pcs/demo/c` directory. Note that pin number 36 and 40 are connected to both terminals of the 100 pF capacitor in the HP 16076A and HP 16356A. To compile and execute the "`cg.c`" program, refer to Chapter 2.

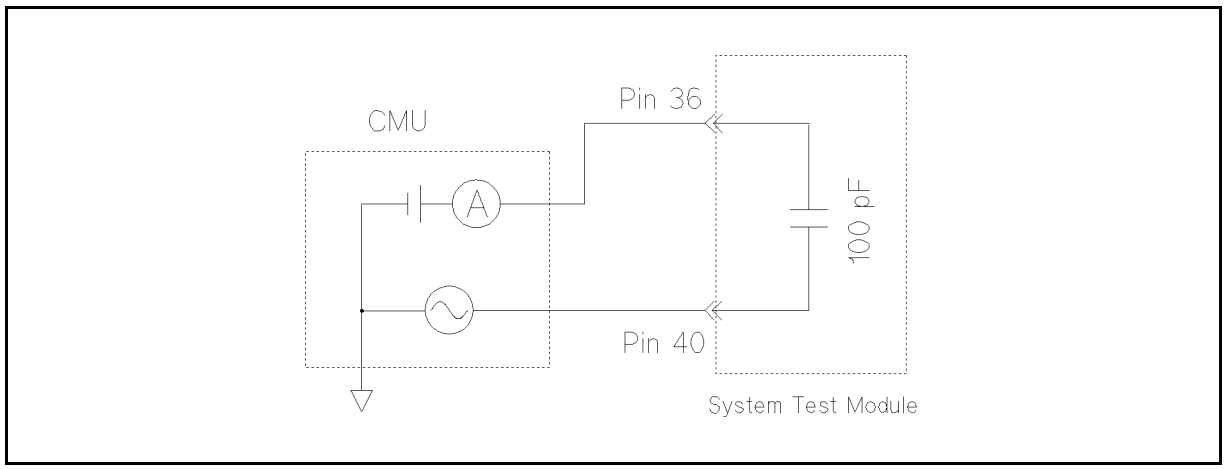


Figure 3-5. Schematic of the Measurement Circuit: "cg.c"

```
1 #include <stdio.h>
2 #include <pcs/tis.h>
3
4 #ifdef __hp9000s700
5 #include <sicl.h>
6 #endif
7
8 main()
9 {
10 #ifdef __hp9000s700
11      INST fd;
12 #else
13      int fd;
14 #endif
15
16     int dummy;
17     double capacitance, conductance, offset;
18
19     if (fd = open_tis_hpib_fd() == -1) error_rep();
20     set_tis_hpib_fd(fd);
21
22     if (init_system() == -1) error_rep();
23     if (lock_system(0) == -1) error_rep();
24
25     if (connect_pin(fncmh, 36) == -1) error_rep();
26     if (connect_pin(fncml, 40) == -1) error_rep();
27
28     printf("Remove the system test module from the SWM\n");
29     printf("Press any key when ready.\n");
30     scanf("%d", &dummy);
31     if (measure_cmu(&offset, &conductance, 1E-10) == -1) error_rep();
32
33     printf("Put the system test module on the SWM\n");
34     printf("Press any key when ready.\n");
35     scanf("%d", &dummy);
36     if (measure_cmu(&capacitance, &conductance, 1E-10) == -1) error_rep();
37
38     if (disable_port(36) == -1) error_rep();
39     if (connect_pin2(0, 36, 40) == -1) error_rep();
40
41     printf("capacitance = %8.5f pF\n", capacitance * 1E12);
42     printf("conductance = %8.5f uS\n", conductance * 1E6);
43 }
44
45 int error_rep()
46 {
47     int errn[2];
48     char errm[100];
49
50     error_info(errn, errm);
51     fprintf(stderr, "%s\n", errm);
52
53     exit(-1);
54 }
```

Figure 3-6. Sample Program for C-G Measurement: "cg.c"

The following is an explanation of the sample program above.

Line 1 to 6	Declare the include files, which are called header files.
Line 19 and 20	Open_tis_hpib_fd searches the SYSCONFIG file to get the session ID, which is set to allow communication with system instruments.

Line 22 and 23	Initialize the system instruments and lock the system instruments to this process.
Line 25 and 26	Connect the CMH port to measurement pin 36 and connect the CML port to measurement pin 40 to connect the CMU to the terminals of the capacitor.
Line 28 to 30	Create a prompt to remove the system test module for the stray capacitance measurement.
Line 31	Measures the stray capacitance.
Line 33 to 35	Creates a prompt to put the system test module on the SWM for the capacitance measurement.
Line 36	Measures the capacitance and conductance using auto ranging.
Line 38	Clears the CMU and sets it to zero output status.
Line 39	Disconnects measurement pins 36 and 40.
Line 45 to 54	Are a subroutine to deal with run-time errors. This subroutine prints the error message on the screen. For more about "`stderr`," see a text book on C programming language.

Forcing Bias Voltage

To force a bias voltage from the CMU, the Force_v function can be used. The usage of the Force_v function is the same as forcing voltage from SMUs. The CMH port address, CMH terminal unit address, or pin number connected to the CMH port is specified as the port. The `force_v()` program line must be used before the `measure_cmu()` program line as shown below:

```
  .  .  .  .  .  .
int err;
  .  .  .  .  .  .
err = force_v(fncmh(), 5.0, 20., 1E-3);
err = measure_cmu(&capacitance, &conductance, 0.0);
  .  .  .  .  .  .
  .  .  .  .  .  .
```

Setting the Measurement Mode

The integration time and test signal level of the CMU can be set using Set_cmu function.

The integration time can be set to SHORT, MEDIUM, or LONG. The test signal level can be set to LOW (10 mVrms) or HIGH (30 mVrms).

For example, to set the integration time to LONG and to set the test signal level to HIGH, the following program lines are available:

```
  .   .   .   .   .   .
int err;
  .   .   .   .   .   .
err = set_cmu(INTEG_LONG, SIG_LEVEL_HIGH);
  .   .   .   .   .   .
  .   .   .   .   .   .
```

General C·G Measurements (CMU84)

This section covers how to measure capacitance and conductance of the DUTs using the CMU84 (HP 4284A 20 Hz–1 MHz Precision LCR Meter). The topics of this section are as follows:

- To measure the capacitance and conductance, use the Measure_cmu84 function.
- To force a bias voltage, use the Force_v function.
- To set integration time and test signal level for the CMU84, use the Set_cmu function.

Note

If the CMU84 is equipped with Option 001 (power amplifier), the CMU84 can force a bias voltage from −40 V to +40 V and can set a signal level from 0 Vrms to 20 Vrms. Otherwise, the CMU84 can force 0 V, 1.5 V, or 2.0 V as a bias voltage and can set a signal level from 0 Vrms to 2.0 Vrms.

Note

Before measuring the capacitance of the DUTs, stray capacitance from the cables and SWM must be measured. After measuring the capacitance of the DUTs, you must subtract the stray capacitance from the measured capacitance. For details, see "Technical Information" in *System Information*.

C·G Measurements

To measure the capacitance and conductance using a CMU84, the Measure_cmu84 function is used. After the measurement, the Disable_port function is used to clear the CMU84.

Figure 3-7 and Figure 3-8 show an example for measuring a capacitance (100 pF) in the HP 16076A or HP 16356A System Test Module. The program named "`cg84.c`" is stored in the `/usr/pcs/demo/c` directory. Note that pin numbers 36 and 40 are connected to both terminals of the 100 pF capacitor in the HP 16076A and HP 16356A. To compile and execute the "`cg84.c`" program, refer to Chapter 2.

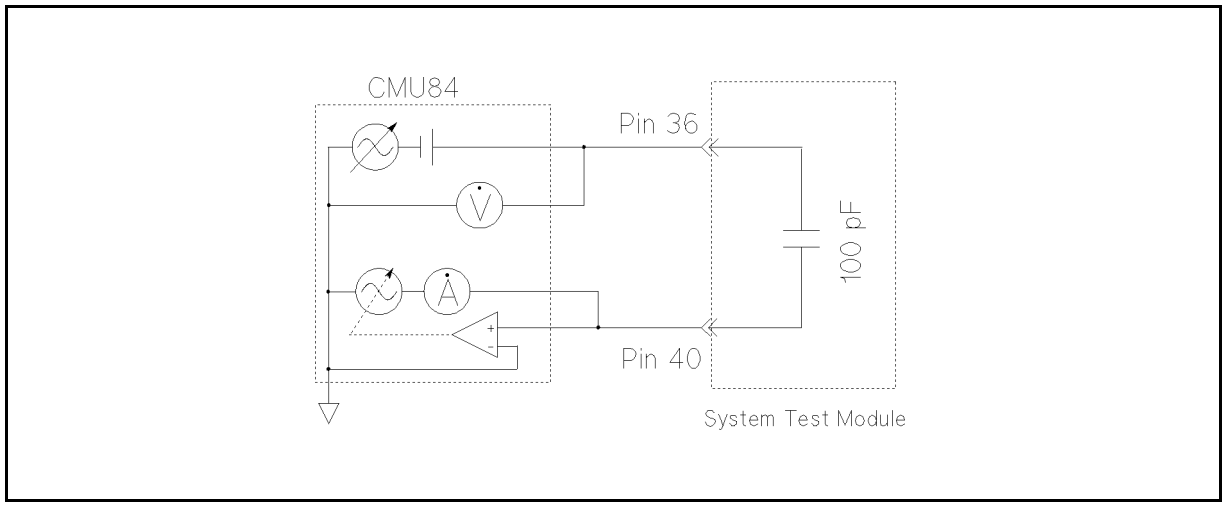


Figure 3-7. Schematic of Measurement Circuit: "cg84.c"

```
1 #include <stdio.h>
2 #include <pcs/tis.h>
3
4 #ifdef __hp9000s700
5 #include <sicl.h>
6 #endif
7
8 main()
9 {
10 #ifdef __hp9000s700
11      INST fd;
12 #else
13      int fd;
14 #endif
15
16     int dummy;
17     double capacitance, conductance, offset;
18
19     if (fd = open_tis_hpib_fd() == -1) error_rep();
20     set_tis_hpib_fd(fd);
21
22     if (init_system() == -1) error_rep();
23     if (lock_system(0) == -1) error_rep();
24
25     if (connect_pin(fncmh, 36) == -1) error_rep();
26     if (connect_pin(fncml, 40) == -1) error_rep();
27
28     printf("Remove the system test module from the SWM\n");
29     printf("Press any key when ready.\n");
30     scanf("%d", &dummy);
31     if (measure_cmu84(&offset, &conductance, 1E-10) == -1) error_rep();
32
33     printf("Put the system test module on the SWM\n");
34     printf("Press any key when ready.\n");
35     scanf("%d", &dummy);
36     if (measure_cmu84(&capacitance, &conductance, 1E-10) == -1) error_rep();
37
38     if (disable_port(36) == -1) error_rep();
39     if (connect_pin2(0, 36, 40) == -1) error_rep();
40
41     printf("capacitance = %8.5f pF\n", (capacitance - offset) * 1E12);
42    printf("conductance = %8.5f uS\n", conductance * 1E6);
43 }
44
45 int error_rep()
46 {
47     int errn[2];
48     char errm[100];
49
50     error_info(errn, errm);
51     fprintf(stderr, "%s\n", errm );
52
53     exit( -1 );
54 }
```

Figure 3-8. Sample Program for C-G Measurement: "cg84.c"

The following is an explanation of the above sample program.

Line 1 to 6 — Declare the include files, which are called header files.

Line 19 and 20 — Open_tis_hpib_fd searches the SYSCONFIG file to get the session ID, which is set to allow communication with system instruments.

Line 22 and 23	Initialize the system instruments and lock the system instruments to this process.
Line 25 and 26	Connect the CMH port to measurement pin 36 and connect the CML port to measurement pin 40 to connect the CMU84 to the terminals of the capacitor.
Line 28 to 30	Creates a prompt to remove the system test module for the stray capacitance measurement.
Line 31	Measures the stray capacitance.
Line 33 to 35	Creates a prompt to put the system test module on the SWM for the capacitance measurement.
Line 36	Measures the capacitance and conductance using auto ranging.
Line 38	Clears the CMU84 and sets it to zero output status.
Line 39	Disconnects measurement pins 36 and 40.
Line 45 to 54	Are a subroutine to deal with run-time errors. This subroutine print error messages on the screen. For more about "`stderr`," see a text book on the C programming language.

Forcing Bias Voltage

To force a bias voltage from the CMU84, the Force_v function can be used. The usage of Force_v function is the same as forcing voltage from SMUs. The CMH port address, CMH84 terminal unit address, or pin number connected to the CMH port is specified as the port. The `force_v()` program line must be used before the `measure_cmu84()` program line, as shown below:

```
  .  .  .  .  .  .
int err;
  .  .  .  .  .  .
err = force_v(fncmh(), 5.0, 20.0, 1E-3);
err = measure_cmu84(&capacitance, &conductance, 0.0);
  .  .  .  .  .  .
  .  .  .  .  .  .
```

Setting the Measurement Mode

The integration time and test signal level of the CMU84 can be set using Set_cmu84 function.

The integration time can be set to SHORT, MEDIUM, or LONG. The test signal level can be set from 0 Vrms to 20 Vrms, if the CMU84 is equipped with Option 001 (power amplifier). Otherwise, the test signal voltage level can be set from 0 Vrms to 2.0 Vrms.

For example, to set the integration time to LONG and to set the test signal level to 1.0 Vrms, the following program lines are available:

```
  .   .   .   .   .   .
int err;
  .   .   .   .   .   .
err = set_cmu84(INTEG_LONG, 1.0);
  .   .   .   .   .   .
  .   .   .   .   .   .
```

Sweep I-V Measurements

This section covers how to program for various sweep I-V measurements. The topics of this section are as follows:

- For staircase sweep measurements, the Set_iv function is used to set the parameters and the Sweep_iv function is used to start the sweep and execute the measurements.
- For synchronous staircase sweep measurements, the Set_sync function is used to set the parameters of the synchronous sweep port. The parameters of the primary sweep port are set by the Set_iv function. The Sweep_iv function is also used to start the synchronous sweep and to execute the measurements.
- For pulsed sweep measurements, the Set_piv function is used to set the parameters. The Sweep_iv function is also used to start the pulsed sweep and to execute the measurements.
- For staircase sweep with pulsed bias measurements, the Set_pbias function is used to set the parameters of the pulsed bias port. The parameters of the primary sweep port are set by the Set_iv function. The Sweep_iv function is also used to start the sweep with pulsed bias and to execute the measurements.
- For multichannel sweep measurements, the Sweep_miv function is used to start the sweep and to execute the measurements at the specified ports.
- For real-time sweep measurements, the Rsweep_iv or Rsweep_miv function is used to start the sweep and to execute the measurements in real-time.

Staircase Sweep Measurements

To perform staircase sweep measurements, the Set_iv and Sweep_iv functions are used. Figure 3-9 shows the typical wave form of the staircase sweep measurements.

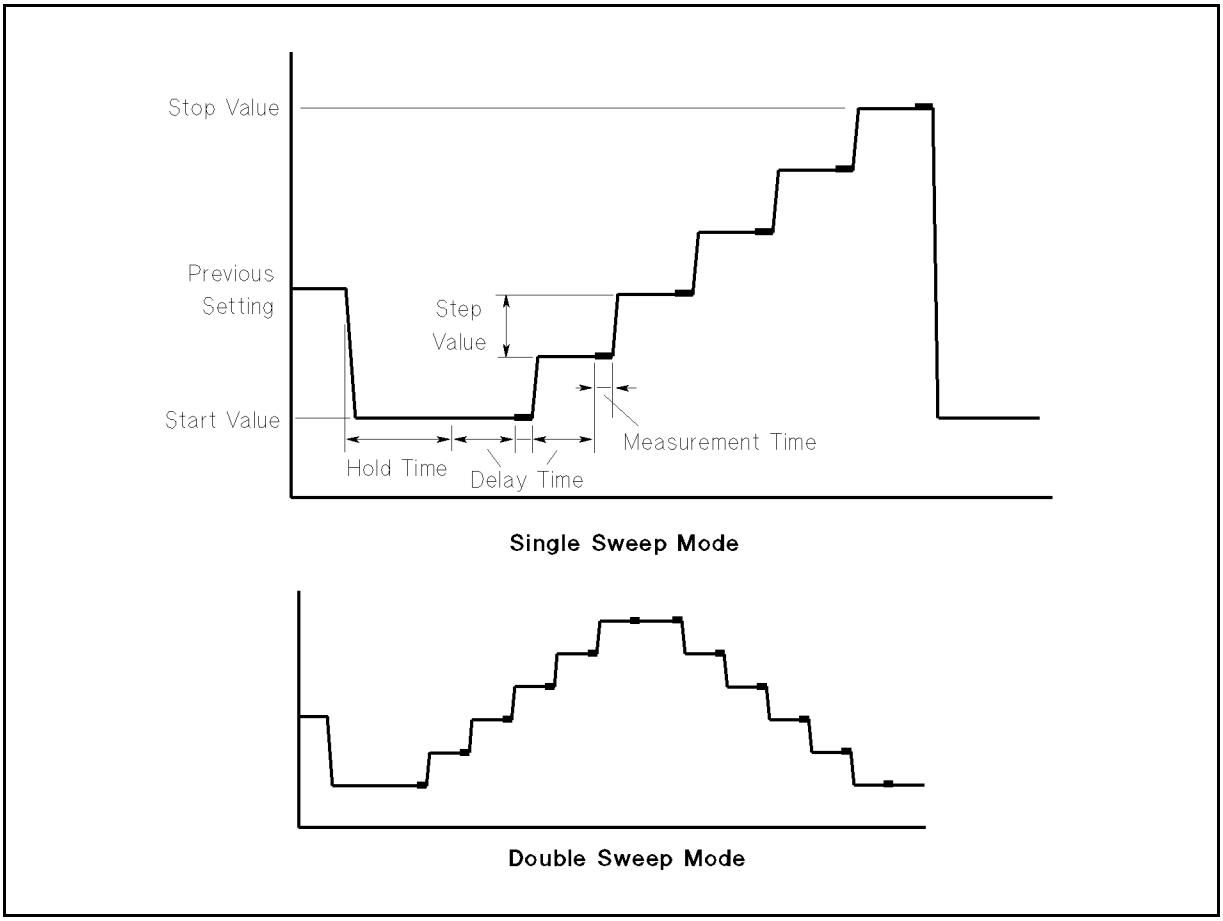


Figure 3-9. Typical Wave Form of Staircase Sweep Measurements

The Set_iv function is used to set the parameters of the staircase sweep measurements. The Sweep_iv function is used to trigger the measurements and to get the measured data. For details on these functions, see *Programming Reference for C Library*.

Figure 3-10 and Figure 3-11 show an example for measuring Id–Vds characteristics of a J-FET (HP part number 1855-0082). The gate-source voltage (Vgs) is set to 0.5 V by the SMU1 and the drain-source voltage (Vds) is swept from 0.0 V to −10 V by the SMU2. Then the values of both Vds and Id are displayed. The Id–Vds curve can be drawn on a piece of paper using the Id–Vds data.

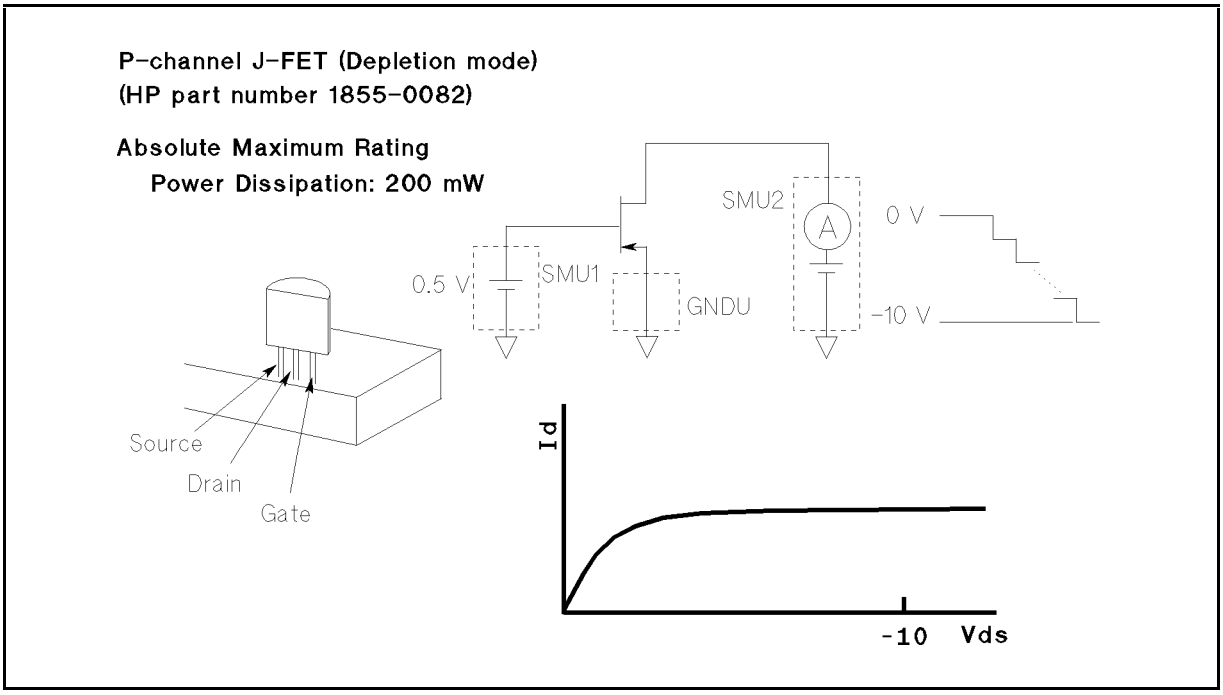


Figure 3-10. Schematic for Example of Id-Vds Characteristics: "sweep.c"

```
1 #include <stdio.h>
2 #include <pcs/tis.h>
3
4 #ifdef __hp9000s700
5 #include <sicl.h>
6 #endif
7
8 main()
9 {
10 #ifdef __hp9000s700
11      INST fd;
12 #else
13      int fd;
14 #endif
15
16     int i;
17     int drain, gate, source;
18     double ic[21], vce[21];
19
20     if (fd = open_tis_hpib_fd() == -1) error_rep();
21     set_tis_hpib_fd(fd);
22
23     if (init_system() == -1) error_rep();
24     if (lock_system(0) == -1) error_rep();
25
26     gate = 8;
27     drain = 12;
28     source = 16;
29
30     if (connect_pin(fnsmu(1), gate) == -1) error_rep();
31     if (connect_pin(fnsmu(2), drain) == -1) error_rep();
32     if (connect_pin(fngnd, source) == -1) error_rep();
33
34     if (set_iv(drain, LINEAR_V, 20.0, 0.0, -10.0, 21, 0.1, 0.05, 1e-2, 0.0, COMP_STOP)
35        == -1) error_rep();
36     if (force_v(gate, 0.5, 2.0, 1e-5) == -1) error_rep();
37     if (sweep_iv(drain, I_MEAS, 0.0, ic, vce, NULL) == -1) error_rep();
38
39     if (disable_port2(gate, drain) == -1) error_rep();
40     if (disconnect_all() == -1) error_rep();
41
42     printf("  Vds (V)     Id (mA)\n");
43     for (i = 0; i < 21; i++)
44         printf("  %6.2f      %7.4f\n", vce[i], ic[i] * 1000);
45 }
46
47 int error_rep()
48 {
49     int errn[2];
50     char errm[100];
51
52     error_info(errn, errm);
53     fprintf(stderr, "%s\n", errm);
54
55     exit(-1);
56 }
```

Figure 3-11. Sample Program for Id-Vds Characteristics: "sweep.c"

The following is an explanation of the above sample program.

Line 1 to 6 Declare the include files, which are called header files.

Line 20 and 21	Open_tis_hpib_fd searches the SYSCONFIG file to get the session ID, which is set to allow communication with system instruments.
Line 22 and 23	Initialize the system instruments and lock the system instruments to this process.
Line 30 to 32	Connect SMU1 port to measurement pin 8 (gate terminal), connect SMU2 port to measurement pin 12 (drain terminal), and connect GNDU port to measurement pin 16 (source terminal).
Line 34	Sets the sweep parameters: linear voltage sweep (0 V to −10 V) using 20 V range at collector terminal.
Line 35	Forces the 0.5 V to the base terminal.
Line 36	Executes the sweep measurements and gets measured data (Ic and Vce).
Line 38	Clears the collector and base terminals and sets it to zero output status.
Line 40	Disconnects the connections of base, collector, and emitter.
Line 46 to 55	Are a subroutine to deal with run-time errors. This subroutine prints error messages on the screen. For more about "`stderr`," see a text book on the C programming language.

Synchronous Staircase Sweep Measurements

To perform synchronous staircase sweep measurements, Set_iv, Set_sync, and Sweep_iv functions are used. In other words, the synchronous sweep port is set to the staircase sweep measurements.

The Set_iv function is used to set the sweep parameters of the primary sweep port. The Set_sync function is used to set the sweep parameter of the synchronous (secondary) sweep port. The Sweep_iv function is used to trigger the measurements and get the measured data. For details on these functions, see the *Programming Reference for C Library*.

For synchronous staircase sweep measurements, the following limitations must be considered:

- The Output mode of the synchronous sweep port is automatically set to the same mode as the primary sweep port. When the primary sweep port is voltage output, the synchronous sweep port is voltage output, and when the primary sweep port is current output, the synchronous sweep port is current output.

- The synchronized output is calculated using the following equation:

  Mode = 0 (synchronizing with positive direction)

  Start2 = *offset* + *ratio* × Start1

  Stop2 = *offset* + *ratio* × Stop1

  Mode = 1 (synchronizing with negative direction)

  Start2 = *offset* − *ratio* × Start1

  Stop2 = *offset* − *ratio* × Stop1

  Where: Start1 and Stop1 are the start and stop values of Set_iv function, and Start2 and Stop2 are the calculated start and stop values of Set_sync function.

- The number of steps is the same as the number of steps in Set_iv function.

Only a Set_sync function is executed between the Set_iv and Sweep_iv functions. Note that more than one synchronous sweep port cannot be specified. A programming example is as follows:

```
  .   .   .   .   .   .
int err;
int number;
double vstart, vstop, hold, delay, comp1;
double offset, ratio, comp2;
double meas1[51], swp1[51], swp2[51];
  .   .   .   .   .   .
  .   .   .   .   .   .
err = set_iv(24, LINEAR_V, 20.0, vstart, vstop, number, hold, delay, comp1,
             0.0, COMP_STOP);
err = set_sync(32, POS_SYNC, offset, ratio, comp2, 0.0);
err = sweep_iv(32, I_MEAS, 0.0, meas1, swp1, swp2);
  .   .   .   .   .   .
  .   .   .   .   .   .
```

Pulsed Sweep Measurements

To perform the pulsed sweep measurements, the Set_piv and Sweep_iv functions are used. Figure 3-12 shows the typical wave form of the pulsed sweep measurements.

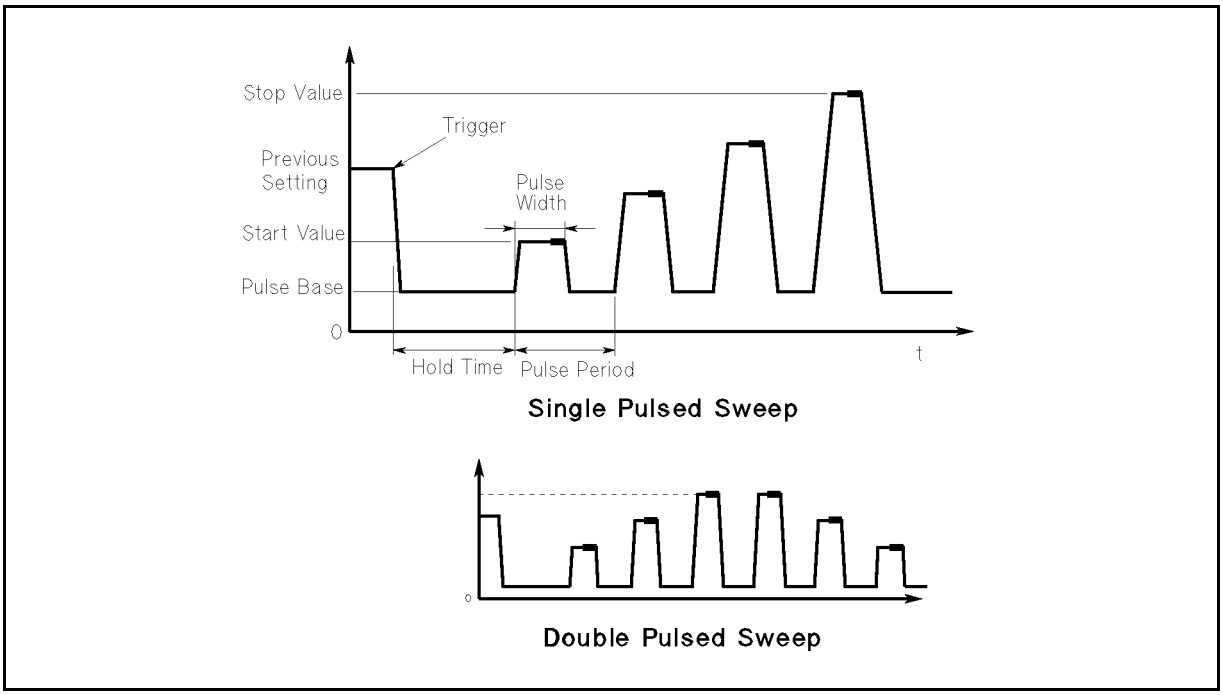


Figure 3-12. Typical Wave Form of Pulsed Sweep Measurements

The Set_piv function is used to set the parameters of pulsed sweep measurements. The Sweep_iv function is used to trigger the measurements and get the measured data. For details on these functions, see *Programming Reference for C Library*.

A programming example is as follows:

```
  .   .   .   .   .   .
int err;
int number;
double vbase, vstart, vstop, width, period, hold, comp;
double meas1[101], swp1[101];
  .   .   .   .   .   .
  .   .   .   .   .   .
err = set_piv(24, LINEAR_V, 20.0, vbase, vstart, vstop, number, width,
              period, hold, comp, COMP_STOP);
err = sweep_iv(32, I_MEAS, 0.0, meas1, swp1, NULL);
  .   .   .   .   .   .
  .   .   .   .   .   .
```

Staircase Sweep with Pulsed Bias Measurements

To perform the staircase sweep with pulsed bias measurements, the Set_iv, Set_pbias, and Sweep_iv functions are used. In other words, a pulsed bias port is added to the staircase sweep measurements.

Figure 3-13 shows a typical wave form of the staircase sweep with pulsed bias measurements.

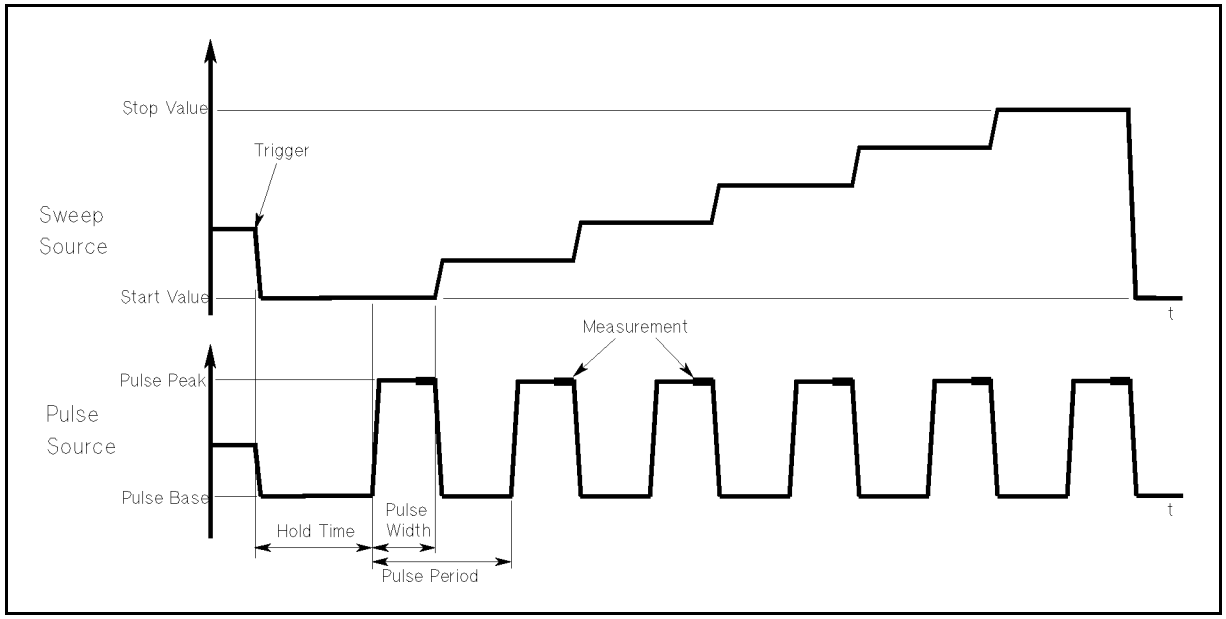


Figure 3-13. Typical Wave Form of Staircase Sweep with Pulsed Bias Measurements

The Set_iv function is used to set the parameters of the staircase sweep port. The Set_pbias function is used to set the sweep parameter of the pulsed bias port. The Sweep_iv function is used to trigger the measurements and get the measured data. For details on these functions, see *Programming Reference for C Library*.

The sweep source is synchronized with the pulsed bias and ignores the hold time and delay time settings of Set_iv function.

A programming example is as follows:

```
  .   .   .   .   .   .
int err;
int number;
double vstart, vstop, hold_dmy, delay_dmy, comp_sweep;
double vbase, vpeak, width, period, hold, comp_bias;
  .   .   .   .   .   .
err = set_iv(24, LINEAR_V, 20., vstart, vstop, number, hold_dmy, delay_dmy,
             comp_sweep, 0., COMP_STOP);
err = set_pbias(28, V_SOURCE, 20., vbase, vpeak, width, period, hold,
                comp_bias);
err = sweep_iv(32, I_MEAS, 0., meas1, swp1, NULL);
  .   .   .   .   .   .
```

Multichannel Sweep Measurements

The multichannel sweep measurements are used to simultaneously perform measurements at more than one port. You can use this measurement method for staircase sweep and synchronous staircase sweep measurements only.

After setting the sweep parameters using the Set_iv and Set_sync functions, the Sweep_miv function is used to trigger the sweep, perform the measurements, and get the measured data. For details on these functions, see the *Programming Reference for C Library*.

A programming example is as follows:

```
  .  .  .  .  .  .
int err;
int number=101;
double vstart, vstop, hold, delay, comp;
int n=3, ports[3];
double ranges[3], measures[3][101], sweep[3];
  .  .  .  .  .  .
  .  .  .  .  .  .
ports[0] = 8; ports[1] = 12; ports[2] = 16;
ranges[0] = 0.0; ranges[1] = 0.0; ranges[2] = 0.0;
  .  .  .  .  .  .
err = force_v(12, 10.0, 10.0, 0.1)
err = force_v(16, 10.0, 10.0, 0.1)
  .  .  .  .  .  .
err = set_iv(8, LINEAR_V, 20.0, vstart, vstop, number, hold, delay,
            comp, 0.0, COMP_STOP);
err = sweep_miv(n, ports, ranges, measures, sweep, NULL);
  .  .  .  .  .  .
  .  .  .  .  .  .
```

The 10 V constant voltage is forced to pin 12 and pin 16. The forced voltage of pin 8 is swept *vstart* to *vstop*, and current measurements are performed simultaneously at pin 8, pin 12, and pin 16. The measured values are stored in the array *measures*.

Real-time Sweep Measurements

The real-time sweep measurements let you get measured data in each step and process the data in real-time. For example, you can draw a characteristic curve on the screen during the measurements. This measurement method is available for four sweep measurements: staircase sweep, synchronous staircase sweep, pulsed sweep, and staircase sweep with pulsed bias measurements.

To perform the sweep measurements in real-time, the Rsweep_iv or Rsweep_miv functions is used instead of the Sweep_iv or Sweep_miv function, respectively. Rsweep_stop is used to stop the real-time sweep measurement. For details on these functions, see the *Programming Reference for C Library*.

The Rsweep_iv function is used to trigger the sweep measurements, perform the measurement at a specified port, and get the measured data in each step. To set the sweep parameters, the Set_iv, Set_piv, Set_sync, or Set_pbias function is used the same as normal sweep measurements.

A programming example is as follows:

```
  .  .  .  .  .  .
int err;
int i, number=101;
double vstart, vstop, hold, delay, comp;
int stat;
double range, meas, sweep;
  .  .  .  .  .  .
  .  .  .  .  .  .
err = set_iv(8, LINEAR_V, 20.0, vstart, vstop, number, hold, delay,
             comp, 0.0, COMP_CONT);
for (i = 0; i < number; i++) {
    err = rsweep_iv(12, I_MEAS, range, &meas, &stat, &sweep, NULL);
    if (err == -1) {
        rsweep_stop();
        break;
    }
    drawing(meas, sweep, stat);
}
  .  .  .  .  .  .
  .  .  .  .  .  .
```

The Rsweep_miv function also lets you trigger the sweep measurements, perform the measurement at specified multichannel ports, and get the measured data in each step.

A programming example follows:

```
  .   .   .   .   .   .
int err;
int i, number=101;
double vstart, vstop, hold, delay, comp;
int n=3, ports[3], stat[3];
double ranges[3], measures[3], sweep;
  .   .   .   .   .   .
  .   .   .   .   .   .
ports[0] = 8; ports[1] = 12; ports[2] = 16;
ranges[0] = 0.0; ranges[1] = 0.0; ranges[2] = 0.0;
  .   .   .   .   .   .
err = force_v(12, 10.0, 10.0, 0.1)
err = force_v(16, 10.0, 10.0, 0.1)
  .   .   .   .   .   .
err = set_iv(8, LINEAR_V, 20.0, vstart, vstop, number, hold, delay,
             comp, 0.0, COMP_CONT);
for (i = 0; i < number; i++) {
    err = rsweep_miv(n, ports, ranges, &measures, &stat, &sweep, NULL);
    if (err == -1) {
        rsweep_stop();
        break;
    }
    drawing(measures[1], sweep, stat[1]);
    drawing(measures[2], sweep, stat[2]);
}
  .   .   .   .   .   .
  .   .   .   .   .   .
```

Search Measurements

This section covers how to program for three search measurements. The topics of this section are as follows:

- For analog search measurements, the Set_asearch function is used to set search parameters and the Search_iv function is used to start the search measurements and get the resultant search value.
- For binary search measurements, the Set_bsearch function is used to set search parameters and the Search_iv function is used to start the search measurements and get the resultant search value. The Set_sync function can be used to set the port synchronized with the search port.
- For linear search measurement, the Set_lsearch function is used to set search parameters and the Search_iv function is used to start the search measurements and get the resultant search value. The Set_sync function can be used to set the port synchronized with the search port.

Analog Search Measurements

To perform the analog search measurements, the Set_asearch and Search_iv functions are used. Figure 3-14 shows the typical wave form of the analog search measurements.

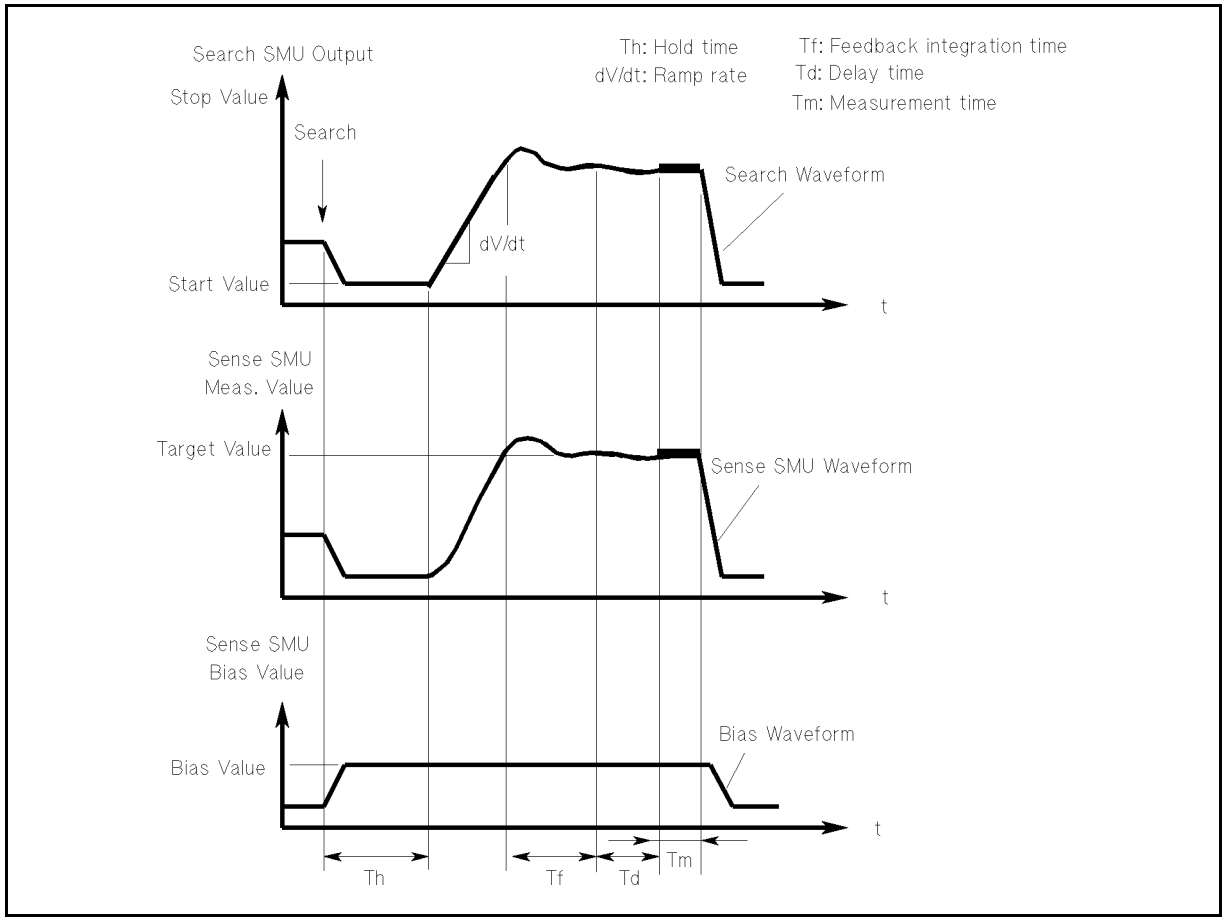


Figure 3-14. Typical Wave Form of Analog Search Measurements

The Set_asearch function is used to set the parameters of the analog search measurements. The Search_iv function is used to trigger the search measurement and get the resultant search value. For more details on these functions, see *Programming Reference for C Library*.

Figure 3-15 and Figure 3-16 show an example for measuring the threshold voltage (Vth) of a J-FET (HP part number 1855-0082). The gate terminal is connected to the search SMU and the drain terminal is connected to the sense SMU. The sense SMU is set to the voltage source forcing −10 V constantly. The voltage output of the search SMU is analogically swept from 0.0 V to 5.0 V and the current value of the sense SMU is monitored. When the monitored value is reached to the target value, the analogue sweeping of the search SMU is complete and the output value is returned.

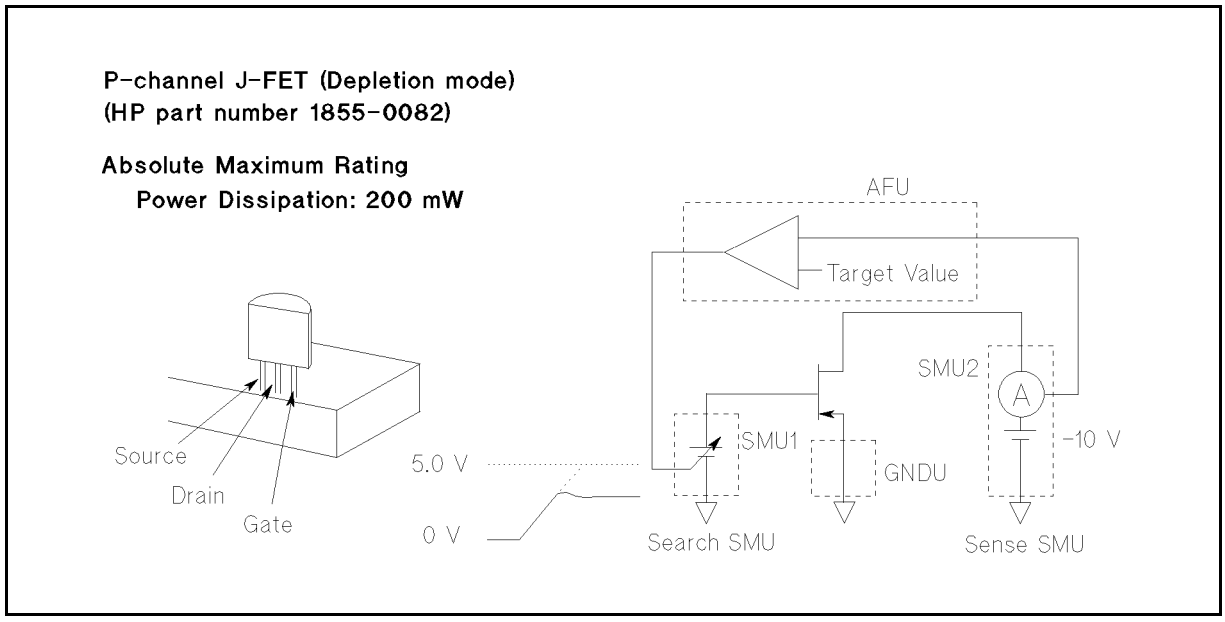


Figure 3-15. Schematics for Example of threshold voltage: "search.c"

```
1 #include <stdio.h>
2 #include <pcs/tis.h>
3
4 #ifdef __hp9000s700
5 #include <sicl.h>
6 #endif
7
8 main()
9 {
10 #ifdef __hp9000s700
11      INST fd;
12 #else
13      int fd;
14 #endif
15
16     int drain, gate, source, stat;
17     double search;
18
19     if (fd = open_tis_hpib_fd() == -1) error_rep();
20     set_tis_hpib_fd(fd);
21
22     if (init_system() == -1) error_rep();
23     if (lock_system(0) == -1) error_rep();
24
25     gate = 8;
26     drain = 12;
27     source = 16;
28
29     if (connect_pin(fnsmu(1), gate) == -1) error_rep();
30     if (connect_pin(fnsmu(2), drain) == -1) error_rep();
31     if (connect_pin(fngnd, source) == -1) error_rep();
32
33     if (set_asearch(gate, drain, NEG_FEEDBACK, VS_IM, 0.0, 5.0, -10.0,
34                     -1e-6, 100.0, 1e-6, --1.15e-6, 0.1, 0.1, 0.1)
35          == -1) error_rep();
36     if (search_iv(&search, &stat, NULL) == -1) error_rep();
37
38     if (disable_port2(gate, drain) == -1) error_rep();
39     if (disconnect_all() == -1) error_rep();
40
41     printf("Status: %d   Vth = %f (V)\n", stat, search);
42 }
43
44 int error_rep()
45 {
46     int errn[2];
47     char errm[100];
48
49     error_info(errn, errm);
50     fprintf(stderr, "%s\n", errm);
51
52     exit(-1);
53 }
```

Figure 3-16. Sample Program for Threshold Voltage: “search.c”

Binary Search Measurements

To perform the binary search measurements, the Set_bsearch and Search_iv functions are used. Figure 3-17 shows the typical wave form of the binary search measurements.

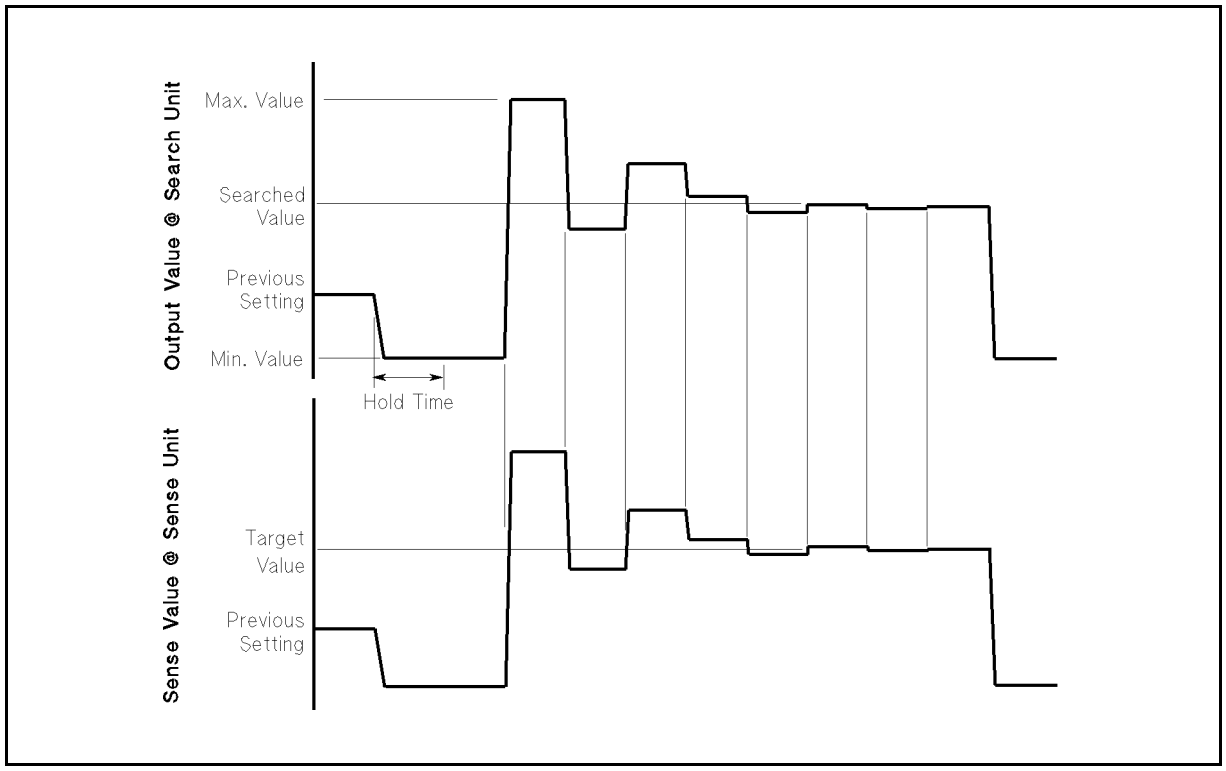


Figure 3-17. Typical Wave Form of Binary Search Measurements

The Set_bsearch function is used to set the parameters of the binary search measurements. The Search_iv function is used to trigger the search measurement and get the resultant search value. Set_sync function is used to set a secondary port synchronized with the search port. The output mode (current or voltage output) is automatically set to the same mode as the primary search port and output value of the synchronous port is determined by the following equations:

Mode = 0

Synchronous port output = *offset* + search port output

Mode = 1

Synchronous port output = *offset* − search port output

Where, the *mode* and *offset* are parameters of Set_sync function.

For more details on these functions, see *Programming Reference for C Library*.

A programming example follows:

```
  .   .   .   .   .   .
int err;
  .   .   .   .   .   .
  .   .   .   .   .   .
err = force_v(drain, bias, 20.0, -0.02);
err = set_bsearch(gate, drain, VS_NORM_ACC, min, max, target, sense_range,
                  search_comp, hold, condition);
err = search_iv(&search, &stat, &sense);
  .   .   .   .   .   .
  .   .   .   .   .   .
```

Linear Search Measurements

To perform the linear search measurements, the Set_lsearch and Search_iv functions are used. Figure 3-18 shows the typical wave form of the linear search measurements.

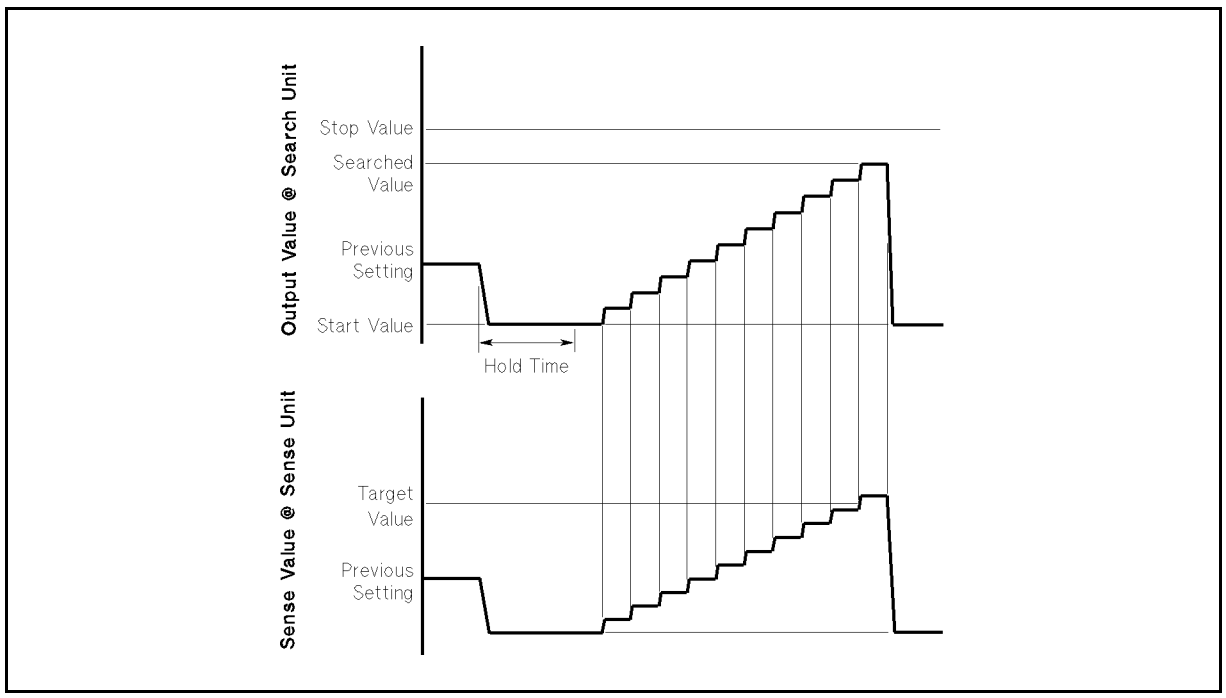


Figure 3-18. Typical Wave Form of Linear Search Measurements

The Set_lsearch function is used to set the parameters of the linear search measurements. The Search_iv function is used to trigger the search measurement and get the resultant search value. The Set_sync function is used to set a secondary port synchronized with search port. The output mode (current or voltage output) is automatically set to the same mode as the primary search port and output value of the synchronous port is determined by the following equations:

Mode = 0

Synchronous port output = *offset* + search port output

Mode = 1

Synchronous port output = *offset* − search port output

Where, *mode* and *offset* are parameters of Set_sync function.

For more details on these functions, see *Programming Reference for C Library*.

A programming example follows:

```
  .  .  .  .  .  .
int err;
  .  .  .  .  .  .
  .  .  .  .  .  .
err = set_lsearch(base, collector, IS_RESET_ABOVE, start, stop, step,
                  target, sense_range, search_comp, hold);
err = search_iv(&ib, &stat, &ic);
  .  .  .  .  .  .
  .  .  .  .  .  .
```

C-V and C-t Measurements (CMU)

This section covers how to program for C-V and C-t measurements using a CMU (HP 4280A 1 MHz C Meter/C-V Plotter). The topics of this section are as follows:

- For C-V measurements, use the Set_cv function to set the parameters and use the Sweep_cv function to start the voltage sweep and execute the capacitance measurements.
- For C-t measurements, use the Set_ct function to set the parameters and use the Sweep_ct function to force a pulsed voltage and execute the capacitance measurements.

Note

Before measuring the capacitance of the DUTs, stray capacitance from the cables and SWM must be measured. After measuring the capacitance of the DUTs, you must subtract the stray capacitance from the measured capacitance. For details, see "Technical Information" in *System Information*.

C-V Measurements

To perform C-V measurements using a CMU, use the Set_cv and Sweep_cv functions. Figure 3-19 shows a typical wave form of the C-V measurements.

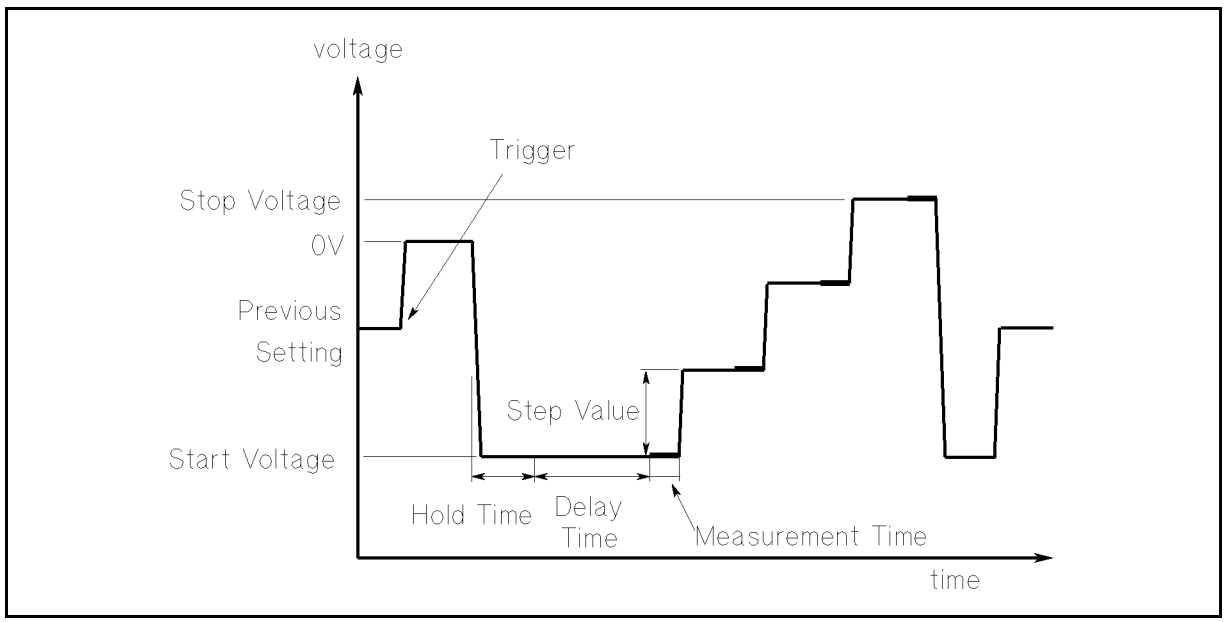


Figure 3-19. Typical Wave Form of C-V Measurements (CMU)

The Set_cv function is used to set the parameters of the C-V measurements. The Sweep_cv function is used to trigger the measurements and get the measured data. For details on these functions, see *Programming Reference for C Library*.

A programming example follows:

```
  .  .  .  .  .  .
int err, number;
double v_start, v_stop, hold, delay;
double range, cap[21], cond[21], bias[21];
  .  .  .  .  .  .
err = set_cv(v_start, v_stop, number, hold, delay);
err = sweep_cv(range, cap, cond, bias);
  .  .  .  .  .  .
```

C-t Measurements

To perform the C-t measurements using the CMU, use the Set_ct and Sweep_ct function. Figure 3-20 shows a typical wave form of the C-t measurements.

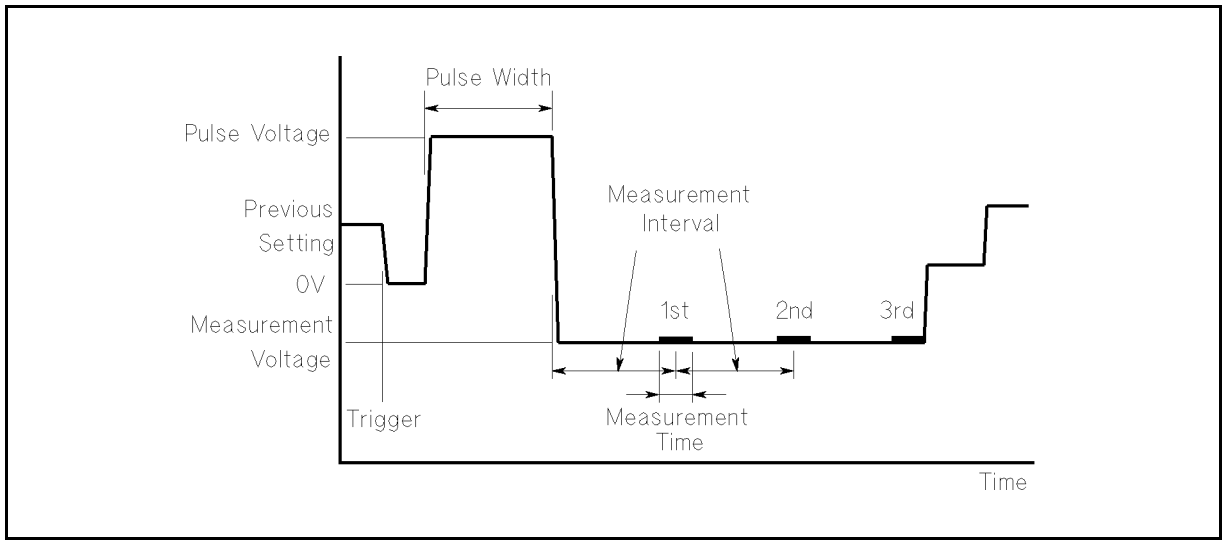


Figure 3-20. Typical Wave Form of C-t Measurements (CMU)

The Set_ct function is used to set the parameters of the C-t measurements. The Sweep_ct function is used to trigger the measurements and get the measured data. For details on these functions, see the *Programming Reference for C Library*.

A programming example is as follows:

```
  .  .  .  .  .  .
int err, number;
double v_pulse, v_meas, number, p_width, interval;
double range, cap[21], cond[21], time[21];
  .  .  .  .  .  .
err = set_ct(v_pulse, v_meas, number, p_width, interval);
err = sweep_ct(range, cap, cond, time);
  .  .  .  .  .  .
```

C-V Measurements (CMU84)

This section covers how to program for C-V measurements using the CMU84 (HP 4284A 20 Hz–1 MHz Precision LCR Meter). The topics of this section are as follows:

- For C-V measurements, use the Set_cv function to set the parameters and use the Sweep_cv84 function to start the voltage sweep and to execute the capacitance measurements.

C-V Measurements

To perform the C-V measurements using the CMU84, the Set_cv84 and Sweep_cv84 functions are used. Figure 3-21 shows a typical wave form of the C-V measurements.

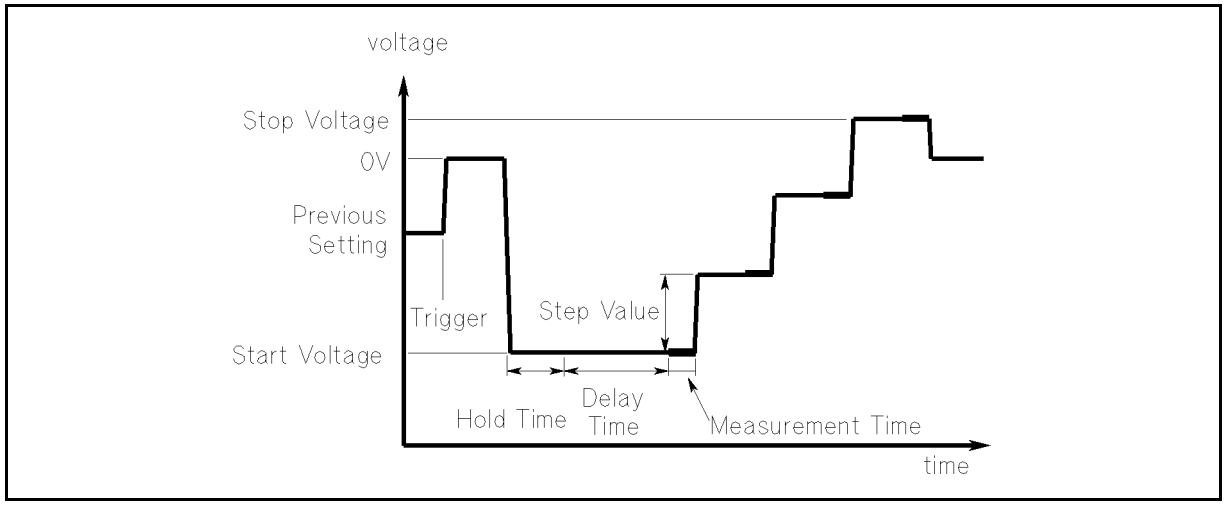


Figure 3-21. Typical Wave Form of C-V Measurements (CMU84)

The Set_cv84 function is used to set the parameters of the C-V measurements. The Sweep_cv84 function is used to trigger the measurements and to get the measured data. For details on these functions, see the *Programming Reference for C Library*.

Note

Before measuring the capacitance of the DUTs, the stray capacitance from the cables and SWM must be measured. After measuring the capacitance of the DUTs, you must subtract the stray capacitance from the measured capacitance. For details, see "Technical Information" in *System Information*.

A programming example follows:

```
  .  .  .  .  .  .
int err, number;
double v_start, v_stop, hold, delay;
double range, cap[21], cond[21], bias[21];
  .  .  .  .  .  .
err = set_cv84(v_start, v_stop, number, hold, delay);
err = sweep_cv84(range, cap, cond, bias);
  .  .  .  .  .  .
```

Program Storage Function with DCS

This section provides information on how to use the DCS program memory functions. For more about the concept of the DCS program memory function, see the *System Information*.

You can make test programs using TIS to measure device parameters with the HP 4062 system. TIS is a powerful command set that controls the HP 4062 system instruments and measures device parameters. You can also use the HP 4062 system with a wafer prober, and measure the device parameters of each chip on each wafer. If you use the looping routine and use TIS functions, such as Force_i or Measure_v, TIS sends the DCS control commands via the HP-IB. Sending the control commands using this method doesn't take much time, but if you measure device parameters for several chips, the total time can become lengthy.

To solve this problem and to increase test throughput, you can use the *DCS program memory* functions. The system controller sends DCS control command sets in the DCS program memory segments before measurements are started. Then the controller sends only a trigger to start a measurement and get measured data. The triggers are sent to the DCS and the measurements are executed until all the chips are measured.

Note that the SWC, CMU, and CMU84 do not have functions to keep instructions in their memory.

To support the DCS program memory function, the PGMLIB library is supplied. The PGMLIB library includes three categories: DCS program storing functions, DCS program storing support functions, and triggering and getting data functions. The following is a list of all the PGMLIB library function names and a brief descriptions of each:

- DCS program storing functions:

Pgm_start	Declares the beginning of storing in the DCS program memory.
Pgm_end	Declares the end of storing in DCS program memory.
Pgm_force_v	Stores the Force_v operation in program memory.
Pgm_force_i	Stores the Force_i operation in program memory.
Pgm_measure_i	Stores the Measure_i operation in program memory.
Pgm_measure_v	Stores the Measure_v operation in program memory.
Pgm_set_asearch	Stores the Set_asearch operation in program memory.
Pgm_search	Stores the Search operation in program memory.
Pgm_set_pbias	Stores the Set_pbias operation in program memory.
Pgm_measure_p	Stores the Measure_p operation in program memory.
Pgm_set_smu	Stores the Set_smu operation in program memory.
Pgm_set_vm	Stores the Set_vm operation in program memory.
Pgm_ch_sw_on	Stores the Ch_sw_on operation in program memory.
Pgm_ch_sw_off	Stores the ch_sw_off operation in program memory.
Pgm_disable_dcs	Stores the DCS disable_port operation in program memory.
Pgm_wait	Stores a specified waiting time operation in the program memory.

- DCS program storing support functions:

Fnaddr4142	Returns the DCS device selector, which consists of a logical unit number and an address bus.
Fnunit4142	Returns a DCS unit address from the DCS port address.
Fnchannel4142	Returns a DCS channel number from a DCS unit address.
Fnrange4142	Returns a DCS range code from a voltage or current value.
Readerror4142	Gets four error codes from the DCS.

- Triggering and getting data functions:

Run_dcs_program	Triggers a specified program stored in the DCS program memory.
Readout_dcs	Gets the measured data and measurement status for the spot, pulsed spot, and search measurements.
Readsweep_dcs	Gets the measured data and measurement status for the sweep measurements.

Note

After measurements are executed using the DCS program memory feature, the operation of TIS functions that control the DCS causes an error because the TIS functions keep the settings of the previous execution in the controller's memory. Previous settings of the program memory segment do not remain in the controller's memory. To recover from the execution of the program memory segment, execute the following functions to reset the DCS:

- Init_system
- Disable_port_all
- Long_cal
- Short_cal

The DCS control commands for binary search and linear search cannot be stored in program memory. The output value of the search unit is calculated from the previously measured value of the sense unit. However, this operation can only be executed from the system controller.

Also, DCS program storing procedures for sweep measurements are not supplied. In one sweep measurement, the time it takes to send control commands is relatively short compared with the time it takes to receive the measured data. If you want to store programs for sweep measurements in the program memory, use the Pcs_hpib_output function to send the DCS control commands for sweep measurements. More details on this technique are provided later in this chapter.

The source program of the PGMLIB library functions is stored in the "`pgmlib.c`" file under the `/usr/pcs/src/libs` directory.

Note

Cycling the power of the DCS clears the program memory.

Storing DCS Control Commands in the DCS Program Memory

You can store a set of DCS control commands in program memory. The programs stored in program memory are numbered from 1 to 99; therefore, the DCS can keep up to 99 programs in the program memory.

The program statements that are used to store a set of DCS control commands must begin with the Pgm_start function and end with the Pgm_end function. The Pgm_start function has one parameter: the program number of the program stored in program memory, which ranges from 1 to 99. If 0 is specified as the parameter for the Pgm_start function, the DCS control commands are not stored in the program memory, but are executed immediately.

The Pgm_*xxxxx* functions are useful for storing the DCS control commands in the DCS program memory. Replace the *xxxxx* with TIS functions that control the DCS, such as "`force_i`." The arguments of the Pgm_*xxxxx* functions are similar to the arguments of the corresponding TIS functions.

Use the Readout_dcs function for measurement execution statements (such as Measure_i function) that require returned variables. The variable name arguments in which measured data are returned are specified in the Readout_dcs function.

Note that you must keep the pin board relays in the dry-switching mode after ending a program stored in program memory. The pin board relays establish pin-port connections of the SWM. If the pin-port connections are changed after the stored program is executed, your program must force the SMUs to the zero output status. To do so, you can use the Pgm_disable_dcs function before the Pgm_end function. The Pgm_disable_dcs function sets the DCS units to the zero output status.

Note

The ordinary TIS functions (such as the Force_i function) cannot be executed between the Pgm_start and Pgm_end functions. If a TIS function is executed, a mismatched data transfer occurs.

Pgm_*xxxxx* functions are supplied for spot and search measurements, but are not supplied for sweep measurements. You can store the DCS control commands for sweep measurements by sending the control commands directly to the HP-IB. See "Storing Sweep Measurement Commands" for details on how to store a program for sweep measurements.

For example, the following program lets you store a program that measures a 4-terminal integrated resistor:

```
  .  .  .  .  .  .  .  .
INST fd;
double voltage = 1.0, vrange = 2.0, icomp = 1.e-2, irange = 1.e-2;
  .  .  .  .  .  .  .  .
if ((fd = open_tis_hpib_fd()) == -1) error_rep();
set_tis_hpib_fd(fd);
  .  .  .  .  .  .  .  .
if (pgm_start(1) == -1) error_rep();
 if (pgm_force_v(fnsmu(1),voltage,vrange,icomp) == -1) error_rep();
 if (pgm_force_v(fnsmu(2),voltage,vrange,icomp) == -1) error_rep();
 if (pgm_force_v(fnsmu(3),voltage,vrange,icomp) == -1) error_rep();
 if (pgm_force_v(fnsmu(4),voltage,vrange,icomp) == -1) error_rep();
 if (pgm_measure_i(fnsmu(1),irange) == -1) error_rep();
 if (pgm_measure_i(fnsmu(2),irange) == -1) error_rep();
 if (pgm_measure_i(fnsmu(3),irange) == -1) error_rep();
 if (pgm_measure_i(fnsmu(4),irange) == -1) error_rep();
 if (pgm_disable_dcs_all() == -1) error_rep();
if (pgm_end() == -1) error_rep();
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
```

If you store a control command set into program memory that specifies the same segment number as another control command set, the new control command set replaces the existing control command set.

Triggering Program Memory Segments and Getting Data

To execute a program memory segment programmed by Pgm_start ... Pgm_end functions, use the Run_dcs_program function. Specify the segment number in the statement. Use the Readout_dcs or Readsweep_dcs function to get the measured data from the DCS. These statements cannot be executed between the Pgm_start and Pgm_end functions.

The typical control sequence is shown as follows:

1. Control the SWM using the Connect_pin functions and make the desired connections.
2. Trigger the program memory segments in the DCS using the Run_dcs_program function and execute the measurements.
3. Get the measured data using the Readout_dcs function.
4. Disconnect the connections in the SWM.

Note

Be careful to dry switch the relays in the SWM. Do not make or break the relays when voltage or current is being forced to the relays.

The following paragraphs cover how to trigger program memory segments and how to get measured data. A programming example is also shown.

Triggering program memory segments

The Run_dcs_program function lets you set up program segment numbers in the argument. The synopsis is as follows:

```
int run_dcs_program(p_no)
int p_no;

int run_dcs_program2(p_no1, p_no2)
int p_no1, p_no2;

int run_dcs_program3(p_no1, p_no2, p_no3)
int p_no1, p_no2, p_no3;
```

The *p_no1*, *p_no2*, and *p_no3* are program segment numbers. Use `run_dcs_program2()` to execute two program segments, and use `run_dcs_program3()` to execute three program segments. These functions return the execution status: 0 for completion without error. The same segment number can be specified more than once as shown in the following example:

```
  .  .  .  .  .  .  .  .
if ( run_dcs_program(3, 3, 2, 2) == -1 ) error_rep();
  .  .  .  .  .  .  .  .
```

In the example above, program memory segment 3 is executed twice and then program memory segment 2 is executed twice, consecutively. After execution, the measured values are kept in the DCS buffer memory in the execution turn until the controller reads them from the DCS buffer memory.

Note

To terminate the program execution, send the INTR signal to the executed process. To do so, press Break on the keyboard for the default terminal configuration. For details on the technical configuration, see the STTY(1) reference page of the *HP-UX reference* manual.

Tip

After executing the Init_system, Long_cal, or Short_cal function, the first Run_dcs_program function sets the output switches of all the SMUs and VSs and executes the specified programs stored in the DCS program memory. The second or later Run_dcs_program function does not recognize the output switch status until the Init_system, Long_cal, or Short_cal function is executed.

Getting the Measured Data

For spot measurements or search measurements, the Readout_dcs function can be used. The measured data are returned to the specified variable and its status is also returned to the next specified variable. The synopsis of this function is as follows:

```
int readout_dcs(data, st)
double *data;
int *st;

int readout_dcs2(data1, st1, data2, st2)
double *data1, *data2;
int *st1, *st2;

int readout_dcs3(data1, st1, data2, st2, data3, st3)
double *data1, *data2, *data3;
int *st1, *st2, *st3;
```

Where *datax* is a measured data variable and *stx* is its corresponding measurement status variable.

For example, use the following statement to get the data obtained by a voltage measurement at an SMU:

```
  .  .  .  .  .  .  .  .
double vmeas;
int stat;
  .  .  .  .  .  .  .  .
if (readout_dcs(&vmeas,&stat) == -1) error_rep();
  .  .  .  .  .  .  .  .
```

As another example, after executing the analog search measurement, you can get the search value, sense value, and measurement statuses as follows:

```
  .  .  .  .  .  .  .  .
double search, sense;
int stat1, stat2;
  .  .  .  .  .  .  .  .
if (readout_dcs(&search, &stat1, &sense, &stat2) == -1) error_rep();
  .  .  .  .  .  .  .  .
```

The Readsweep_dcs function can be used for sweep measurements. The measured data is transferred in chronological order and returned to the specified array. The measurement statuses corresponding to the measured data are also returned to the specified array. The synopsis of this function is as follows:

```
int readsweep_dcs(n, data, st)
double data[];
int n, st[];

int readsweep_dcs2(n, data1, st1, data2, st2)
double data1[], data2[];
int n, st1[], st2[];

int readsweep_dcs3(n, data1, st1, data2, st2,  data3, st3)
double data1[], data2[], data3[];
int n, st1[], st2[], data3[];

int readsweep_dcs_m(m, n, data, st)
double data[m][n];
int m, n, st[m][n];
```

Where *datax* is a measurement data array name and *stx* is a status array name in which the measurement status is returned. *N* is the step number of the sweep measurement and *m* is the number of measurement ports executed at a time.

For example, when the staircase sweep measurement to draw an Ic-Vc curve is executed, you can get the Ic for each Vc step as follows:

```
  .  .  .  .  .  .  .  .
double ic[101], vc[101];
int n = 101, ic_st[101], vc_st[101];
  .  .  .  .  .  .  .  .
if (readsweep_dcs2(n, ic, ic_st, vc, vc_st) == -1 ) error_rep();
  .  .  .  .  .  .  .  .
```

For details on the data records, see "Binary Measurement Data Output Format" in the HP 4142B *HP-IB Command Reference Manual*.

You can also execute TIS functions that control the SWM, CMU, and CMU84. For example, a capacitance measurement is available using the Measure_cmu function during a measurement using the program memory, as follows:

```
  .  .  .  .  .  .  .  .
double cap, cond, c_range, i_meas;
int stat;
  .  .  .  .  .  .  .  .
if (run_dcs_program(2) == -1 ) error_rep();
if (readout_dcs(&i_meas, &stat) == -1 ) error_rep();
  .  .  .  .  .  .  .  .
if (measure_cmu(&cap, &cond, c_range) == -1 ) error_rep();
  .  .  .  .  .  .  .  .
```

Controlling the SWM

When you execute the program memory segments, you must dry switch the relays in the SWM. If a voltage or current is forced, making or breaking relays results in poor contact of the relays and you will not get reliable measured data.

To keep from damaging the relays, no voltage or current should be forced when the relays make or break. The following programming notes can keep the relays dry switching and can help prevent the relays from being damaged:

- The Connect_pin functions must be executed before the Run_dcs_program functions.
- The Pgm_disable_dcs functions is executed before the Pgm_end functions, and is executed to set the SMUs or VSs to the zero output status.

The following programming example shows how to connect the proper pins and ports:

```
  .  .  .  .  .  .  .  .
double ic, ib;
int ic_st, ib_st;
  .  .  .  .  .  .  .  .
if (pgm_start(4)) == -1) error_rep();
  .  .  .  .  .  .  .  .
 if (pgm_disable_dcs_all() == -1) error_rep();
if (pgm_end() == -1) error_rep();
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
if (connect_pin(fnsmu(1), 24) == -1) error_rep();
if (connect_pin(fnsmu(2), 26) == -1) error_rep();
if (connect_pin(fngnd(), 28) == -1) error_rep();
if (run_dcs_program(4) == -1) error_rep();
if (readout_dcs(&ic, &ic_st, &ib, &ib_st) == -1) error_rep();
if (disconnect_all() == -1) error_rep();
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
```

Programming Example

Figure 3-22 provides an example program measuring sheet resistance and h_{FE} in the chips of a wafer specified in the WAFER1 file. For the W_get, W_start_number, W_move_next functions, see "Probing Control Library (PCL)" in Chapter 6.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/wafer.h>

#define R1      4
#define R2      8
#define TR_B    12
#define TR_C    16
#define TR_E    20

main()
{
    char *ppd_file= "WAFER1";
    int chip, st1, st2;
    double v_r = 1.0;
    double i_r;
    double v_start = 0.0;
    double v_stop = 3.0;
    double v_ce = 10.0;
    double i_c = 1.e-3;
    double i_b;

/********************** Storing DCS programs ***********************/
    if (pgm_start(1) == -1) error_rep();
        if (pgm_force_v(fnsmu(1), v_r, 2.0, 1.e-2) == -1) error_rep();
        if (pgm_measure_i(fnsmu(1)) == -1) error_rep();
        if (pgm_disable_dcs(fnsmu(1)) == -1) error_rep();
    if (pgm_stop() == -1) error_rep();

    if (pgm_start(2) == -1) error_rep();
        if (pgm_set_asearch(fnsmu(1), fnsmu(2), NEG_FEEDBACK, IS_IM_MEAS,
                            v_start, v_stop, v_ce, i_c, 500.0, 100.e-6,
                            i_c * 1.15) == -1) error_rep();
        if (pgm_search_iv() == -1) error_rep();
        if (pgm_disable_dcs2(fnsmu(1), fnsmu(2)) == -1) error_rep();
    if (pgm_stop() == -1) error_rep();

/******************* Starting main test routine ********************/
    if (init_system() == -1) error_rep();
    if (w_get(ppd_file) == -1) error_rep();
    if (w_start_number(1, 1) == -1) error_rep();
    for (chip = 1; chip <= w_total_chip() + 1; chip++) {
        printf("Chip No. : %d\n", chip);
        if (connect_pin(fngnd(), R2) == -1) error_rep();
        if (connect_pin(fnsmu(1), R1) == -1) error_rep();
        if (run_dcs_progam(1) == -1) error_rep();
        if (readout_dcs(i_r, st1) == -1) error_rep();
        printf("  R   = %8.2f (status = %d)\n", v_r/i_r, st1);
        if (connect_pin2(0, R1, R2) == -1) error_rep();
```

Figure 3-22. Sample Program: DCS Program Memory Function (1 of 2)

```
        if (connect_pin(fngnd(), TR_E) == -1) error_rep();
        if (connect_pin(fnsmu(1), TR_B) == -1) error_rep();
        if (connect_pin(fnsmu(2), TR_C) == -1) error_rep();
        if (run_dcs_progam(2) == -1) error_rep();
        if (readout_dcs2(i_b, st1, i_c, st2) == -1) error_rep();
        printf("  hFE = %8.2f (status = %d)\n", i_c/i_b, st2);
        if (connect_pin3(0, TR_C, TR_B, TR_E) == -1) error_rep();
        if (w_move_next() == -1) error_rep();
    }
}
void error_rep()
{
    fprintf(stderr, "tis_errno: %d\n", tis_errno);
    fprintf(stderr, "p_errno:   %d\n", p_errno);
    fprintf(stderr, "w_errno:   %d\n", w_errno);
    exit(-1);
}
```

Sample Program: DCS Program Memory Function (2 of 2)

Optimizing the DCS Program

Optimize your program by writing or rewriting the program using the HP-IB control commands of the DCS. It is beneficial to know the HP-IB control commands of the DCS, which are provided in the HP 4142B manuals.

Programs using the Pgm_*xxxx* functions are not always the most efficient programs. The most efficient program controls the fewest DCS units and changes the least amount of parameters. Note that the HP 4062UX system software cannot optimize your program automatically. You must optimize your program yourself.

For example, the following program is functionally identical to the program shown previously in this chapter that stores an integrated resistor (4 circuit) measurement:

```
  .  .  .  .  .  .  .  .
INST dcs_eid;
int dce_sc, dcs_ba;
char *commands;
  .  .  .  .  .  .  .  .
dce_sc = fnaddr4142() / 100;
dce_ba = fnaddr4142() % 100;
  .  .  .  .  .  .  .  .
if (pgm_start(1) == -1) error_rep();
  if (dcs_eid = pcs_hpib_open(dce_sc)) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "AV-4", 0) == -1) error_rep();
  commands = "DV1,11,1,0.01";
  if (pcs_hpib_output(dcs_eid, dcs_ba, commands, 0) == -1) error_rep();
  commands = "DV3,11,1,0.01";
  if (pcs_hpib_output(dcs_eid, dcs_ba, commands, 0) == -1) error_rep();
  commands = "DV4,11,1,0.01";
  if (pcs_hpib_output(dcs_eid, dcs_ba, commands, 0) == -1) error_rep();
  commands = "DV5,11,1,0.01";
  if (pcs_hpib_output(dcs_eid, dcs_ba, commands, 0) == -1) error_rep();
  commands = "MM1,3,4,5";
  if (pcs_hpib_output(dcs_eid, dcs_ba, commands, 0) == -1) error_rep();
  commands = "XE";
  if (pcs_hpib_output(dcs_eid, dcs_ba, commands, 0) == -1) error_rep();
  commands = "DZ";
  if (pcs_hpib_output(dcs_eid, dcs_ba, commands, 0) == -1) error_rep();
if (pgm_end() == -1) error_rep();
  .  .  .  .  .  .  .  .
```

The `fnaddr4142()` function returns the device selector showing the logical unit number and bus address of the DCS. Four SMUs force 1.0 V (current compliance: 10 mA) to the four resistors. In the example above, the DCS slot configuration is assumed as follows:

SMU1 Slot1: HP 41421B 100 V/100 mA SMU
SMU2 Slot3: HP 41420A 200 V/1 A SMU (occupying two slots)
SMU3 Slot4: HP 41421B 100 V/100 mA SMU
SMU4 Slot5: HP 41421B 100 V/100 mA SMU

In the above program, the following procedures are stored in program memory segment 1:

1. Integration time of the DCS is set to four periods of the line frequency.
2. Four SMUs are set to force 1.0 V with 10 mA current compliance.
3. The current measurement is executed in the four SMUs.
4. When the measurement is finished, the output voltage for the four SMUs is set to 0 V.

The `pcs_hpib_open()` and `pcs_hpib_output()` are used to send the raw ASCII commands to the HP-IB. For details about the `pcs_hpib_open()` and `pcs_hpib_output()`, see "Controlling Unsupported Instruments" or the *Programming Reference for C Library* manual.

Note

HP-IB control commands must be sent to the DCS between Pgm_start and Pgm_end function. If HP-IB control commands of the DCS are sent using Pcs_hpib_output function outside of the Pgm_start and Pgm_end context, a mismatched data transfer occurs.

Note

You must take care of the output switch status of the SMUs or VSs when using the `CN` and `CL` commands. When the output switch is set to off, forcing voltage or current causes an error. It takes about 100 ms to execute the `CN` or `CL` command.

To increase the measurement speed, use the `CN` and `CL` commands as few times as possible.

Note that you do not need to set the output switch status of the SMUs or VSs when using TIS functions, because the TIS functions automatically control the output switches.

Storing Sweep Measurement Commands

To store the DCS control commands for sweep measurements, you can send the DCS control commands directly with the Pcs_hpib_output function. The Pcs_hpib_output function must be programmed within the Pgm_start ... Pgm_end context.

Programming with the DCS control commands directly lets you create very useful and diverse test procedures. For details on how to program using DCS control commands (ASCII commands), see the HP 4142B *Operation Manual*.

Note

Even when a sweep measurement program stored in program memory is executed, the total throughput may not increase. The transfer time of a sweep measurement program does not consume a lot of the total measurement time.

The following example illustrates a measurement of the Ic-Vce characteristics of a bipolar transistor. A staircase voltage (Vce) is forced from 0 V to 10 V in 101 steps to the collector terminal, and collector current (Ic) is measured for four base current values (0.5 μA, 1.5 μA, 2.5 μA, and 3.5 μA).

```
  .  .  .  .  .  .  .  .  .
INST dcs_eid;
int dce_sc, dcs_ba;
int ic_st[101];
double ic[101];
  .  .  .  .  .  .  .  .  .
dce_sc = fnaddr4142() / 100;
dce_ba = fnaddr4142() % 100;
  .  .  .  .  .  .  .  .  .
if (pgm_start(4) == -1) error_rep();
  if (dcs_eid = pcs_hpib_open(dce_sc)) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "WV3,1,1,12,0,10,101,0.2", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "WT0.01", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "MM2,3", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "DI1,1,15,5E-7,10", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "XE", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "DI1,1,15,1.5E-6,10", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "XE", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "DI1,1,15,2.5E-6,10", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "XE", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "DI1,1,15,3.5E-6,10", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "XE", 0) == -1) error_rep();
  if (pcs_hpib_output(dcs_eid, dcs_ba, "DZ", 0) == -1) error_rep();
if (pgm_end() == -1) error_rep();
  .  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .
if (connect_pin(fngnd(), 16) == -1) error_rep();
if (connect_pin(fnsmu(2), 20) == -1) error_rep();
if (connect_pin(fnsmu(1), 24) == -1) error_rep();
if (run_dcs_program(4) == -1) error_rep();
if (connect_pin3(0, 24, 20, 16) == -1) error_rep();
if (readsweep_dcs(ic, ic_st) == -1) error_rep();
draw_graph(101, 5.e-7, ic, ic_st);
if (readsweep_dcs(ic, ic_st) == -1) error_rep();
draw_graph(101, 1.5e-6, ic, ic_st);
if (readsweep_dcs(ic, ic_st) == -1) error_rep();
draw_graph(101, 2.5e-6, ic, ic_st);
if (readsweep_dcs(ic, ic_st) == -1) error_rep();
draw_graph(101, 3.5e-6, ic, ic_st);
  .  .  .  .  .  .  .  .  .
```

In the example above, the following is assumed:

- Transistor terminal configuration

Emitter	Pin 16
Collector	Pin 20
Base	pin 24

- DCS slot configuration

SMU1	Slot 1: HP 41421B 100 V/100 mA SMU
SMU2	Slot 3: HP 41420A 200 V/1 A SMU

- The synopsis of the `draw_graph()` function is as follows:

```
void draw_graph(num, ib, ic, ic_st)
int num;                          the step numbers of sweep measurement
double ib;                        base current value
double ic[];                      pointer of array including collector current
int ic_st[];                      pointer of array including collector status
```

DCS Control Commands:

`WV`	Specifies the voltage staircase sweep source and its parameters.
`WT`	Sets the hold time and delay time for staircase sweep measurements.
`MM2`	Sets the measurement mode to staircase sweep measurement, specifying the measurement units.
`DI`	Forces output current from the specified unit.
`XE`	Triggers the DCS to start the measurements.
`DZ`	Sets the specified units to zero output.

The `draw_frame()` function creates a frame of the Ic-Vce characteristics graph. The `draw_graph()` function draws lines on the frame created by the `draw_frame()` function.

Controlling Unsupported Instruments

To control the system instruments (DCS, SWM, and CMU or CMU84), the Test Instruction Set (TIS) functions are provided. But functions are not provided to control unsupported instruments. You must program by yourself to control the unsupported instruments. Note that it is assumed that the unsupported instruments (called HP-IB instruments in this section) are equipped with HP-IB or GP-IB interface.

The HP 4062UX system software provides the low level functions to control the HP-IB as follows:

Pcs_hpib_open	Opens an HP-IB interface.
Pcs_hpib_output	Sends a string of characters to HP-IB instruments.
Pcs_hpib_enter	Receives a string of characters from HP-IB instruments.
Pcs_hpib_clear	Clears the HP-IB instruments.
Pcs_hpib_spoll	Performs serial polling on HP-IB instruments.

For more information on these functions, see *Programming Reference for C Library*. The source programs of these functions are stored in the `/usr/pcs/src/libs/pgmlib.c` file. You can also refer to the *HP Standard Instrument Control Library User's Guide and Reference* manuals.

The following paragraphs show how to program to send or receive for HP-IB instruments, such as a digital multimeter.

Note

To communicate with HP-IB instruments by the Pcs_hpib_*xxxx* functions, logical unit numbers must be set in the /usr/pil/etc/hwconfig.cf file and the /usr/pcs/etc/SYSCONFIG file. For details about how to set the logical unit numbers, see Chapter 5 in the *HP 4062UX/F Installation Manual*.

Opening an HP-IB Interface

Before your controller communicates to the HP-IB instruments, the HP-IB interface must be open. This procedure is done as follows:

```
#include <pcs/tis.h>
#include <sicl.h>
  .   .   .   .   .   .   .   .
INST eid;
  .   .   .   .   .   .   .   .
eid = pcs_hpib_open( sc );
  .   .   .   .   .   .   .   .
```

Where *sc* shows the logical unit number of the HP-IB interface. The Pcs_hpib_open function returns the session ID (eid) for the opened HP-IB interface.

If any errors occur, `pcs_hpib_open()` returns −1 and `errno` includes the error number.

A closing procedure for the opened HP-IB interface is not needed. After program execution is complete or terminated, the eid for the opened HP-IB interface is automatically deleted and communication between the controller and HP-IB instruments ends.

Sending Control Commands to the Instruments

To send control commands to the HP-IB instruments, use the Pcs_hpib_output function. The Pcs_hpib_output function is similar to the OUTPUT statement of HP BASIC. The synopsis is as follows:

```
int pcs_hpib_output(eid, addr, str, flag)
INST eid;
int addr, flag;
char *str;
```

The *eid* is the session ID returned by the Pcs_hpib_open function. The *addr* is the bus address of an HP-IB instrument. The *str* is a pointer of a string, which is a string of control commands for the HP-IB instruments. The string must be terminated with null character ('\0'). The *flag* defines the terminator of the string as follows:

flag	**Termination**
0	$\cdots$ *commands* $\cdots$ <CR> <LF>
1	$\cdots$ *commands* $\cdots$ with EOI line
2	$\cdots$ *commands* $\cdots$

If no errors occur, the Pcs_hpib_output function returns 0. If any errors occur, the Pcs_hpib_output function returns -1 and `errno` includes the error number.

For example, to send a string to an instrument (HP-IB address: 7)

```
#define DMM 7 /* Digital Multimeter bus address: 7 */
  .   .   .   .   .   .   .   .
INST eid;
  .   .   .   .   .   .   .   .
if ((eid = pcs_hpib_open(8)) == -1) error_rep();
  .   .   .   .   .   .   .   .
if (pcs_hpib_output(eid, DMM, "F2 R3 Z0", 1) == -1) error_rep();
  .   .   .   .   .   .   .   .
```

Receiving Data From the Instruments

To receive data from the HP-IB instruments, use the Pcs_hpib_enter function. The Pcs_hpib_enter function is similar to the ENTER statement of HP BASIC. The synopsis is as follows:

```
int pcs_hpib_enter(eid, addr, str, n)
INST eid;
int addr, n;
char *str;
```

The *eid* is the session ID returned by the Pcs_hpib_open function. The *addr* is the bus address of an HP-IB instrument. The *str* is a pointer to a data buffer where the received data string is stored. The received string includes a termination character and a null character ('\0') is added at the end of the received string. The *n* is the size of the received data buffer. The *n* must be less than the size of the data buffer because of the null character.

If no errors occur, the Pcs_hpib_enter function returns the number of data bytes stored in the *str* successfully. If any errors occur, the Pcs_hpib_enter function returns −1 and `errno` includes the error number.

The following shows an example to receive data from an instrument (HP-IB address: 7):

```
#define DMM 7 /* Digital Multimeter bus address: 7 */
  .   .   .   .   .   .   .   .
INST eid;
int len;
char meas_data[129];
  .   .   .   .   .   .   .   .
if ((eid = pcs_hpib_open(8)) == -1) error_rep();
  .   .   .   .   .   .   .   .
if (pcs_hpib_output(eid, DMM, "F2 R3 Z0", 1) == -1) error_rep();
if ((len = pcs_hpib_output(eid, DMM, meas_data, 128)) == -1) error_rep();
  .   .   .   .   .   .   .   .
```

4

Program Debugging

Introduction

This chapter describes the following measurement program debugging capabilities.

- Error handling
- Port status function
- Monitoring with VFP
- Offline debugging

Error Handling

This section covers how to program for handling the errors that occur while running the program. It is important to know what type of errors occur while running the program, and this facility enables you to reduce the time of programming debugging.

In "Variables and Function for the Error Number", four variables and one function for the error numbers are explained.

In "Useful Macros for Error Handling", useful macro definitions for TIS, PCL, FCL, and prober drivers are introduced.

Variables and Function for the Error Number

The HP-UX and HP 4062UX system provides the following variables and a function in which the error number is stored when an error occurs while running the program:

errno — Error number of the system call. To use this variable, the `errno.h` file must be included with the `#include` in the source program, or the `errno` must be declared as an external integer variable. For the *errno* meanings, see the ERRNO(2) manual pages.

tis_errno — TIS error number that occurs during the TIS routines. To use this variable, the `pcs/tis.h` file must be included with the `#include` in the source program.

p_errno — Error number of the prober drivers that occurs during the prober driver routines. To use this variable, `pcs/prober.h` file must be included with the `#include` in the source program.

w_errno — PCL and FCL error number that occurs during the PCL or FCL routines. To use this variable, the `pcs/wafer.h` file must be included with the `#include` in the source program.

igeterrno() Returns error number of the SICL. To use this function, the sicl.h file must be included with `#include` in the source program.

A programming example follows:

```
#include <stdio.h>
#include <errno.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
  .  .  .  .  .  .  .
  .  .  .  .  .  .  .
    if (force_v( .  .  .  .  . ) == -1) error_rep();
  .  .  .  .  .  .  .
    if (measure_i( .  .  .  .  . ) == -1) error_rep();
  .  .  .  .  .  .  .
}

error_rep()
{
    fprintf(stderr, "tis_errno = %d\n", tis_errno);
    fprintf(stderr, "p_errno   = %d\n", p_errno);
    fprintf(stderr, "w_errno   = %d\n", w_errno);
    fprintf(stderr, "errno     = %d\n", errno);
    fprintf(stderr, "ierrno    = %d\n", igeterrno());
    exit(-1);
}
```

Useful Macros for Error Handling

The TIS, PCL, FCL, and prober drivers of the HP 4062UX system software also provide useful macro definitions, which correspond to all the TIS, PCL, FCL, and prober driver functions. In the header file, uppercase function names are defined to call the functions and then to check the error occurrence. The spelling of the uppercase function name is the same as the spelling of the original function name, which is written in lowercase characters.

For instance, the `CONNECT_PIN` macro is defined in the `pcs/tis.h` file as follows:

```
#define CONNECT_PIN(port,pin) \
        if ( (connect_pin(port,pin)) == -1 ) \
                tis_error_handler(__FILE__, __LINE__, \
                "CONNECT_PIN(port,pin)")
```

Note that the backslash (\) is used to continue to the next line.

The `__FILE__` and `__LINE__` are predefined macros of the C preprocessor. The `__FILE__` is replaced with the source file name and the `__LINE__` is replaced with the line number when source program is compiled.

The Tis_error_handler function used in the error handling macros has the following functions:

- Displays the file name and line number where an error occurs.
- Displays the error number and its error message.
- Displays the name of the function causing the error.
- Aborts the program execution.

When a program that includes error handling macros is compiled and executed, an error occurrence causes messages to be displayed, as shown in the following example:

```
spot.c(14): TIS ERROR 34  Pin board not present.
Error call: CONNECT_PIN(fnsmu(1),12)
```

For prober drivers, the Prober_error_handler function is used to process errors in the same way as the Tis_error_handler function. For PCL and FCL, the Wafer_error_handler function is used.

Note

When these macros are written in a source program text, the left parenthesis ("(") must be placed next to the macro name with no space.

Note

To open the HP-IB data path for communicating with system instruments, use `open_tis_hpib_fd()` and `set_tis_hpib_fd()`. However, you can use the `OPEN_TIS_HPIB` macro instead of these two functions. The `OPEN_TIS_HPIB` macro is defined in the header file `pcs/tis.h` as follows:

```
#define OPEN_TIS_HPIB \
   { INST fd; \
        if ( (fd = open_tis_hpib_fd()) == -1 ) \
        tis_error_handler(__FILE__, __LINE__, "OPEN_TIS_HPIB"); \
     set_tis_hpib_fd(fd); }
```

Figure 4-1 shows a programming example using error handling macros:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <sicl.h>

main()
{
    double current;

    OPEN_TIS_HPIB;
    INIT_SYSTEM;
    LOCK_SYSTEM(0);

    CONNECT_PIN(fnsmu(1), 12);
    CONNECT_PIN(fngnd(), 16);

    FORCE_V(12, 0.5, 2.0, 1E-3);
    MEASURE_I(12, &current, 1E-3);

    DISABLE_PORT(12);
    CONNECT_PIN2(0, 12, 16);

    printf("current = %8.5f (at %8.5f volt)\n", current, 0.5);
}
```

Figure 4-1. A Programming Example: Error Handling Macros

The Tis_error_handler function can also be modified, if needed. When you create the original Tis_error_handler function, use the following template:

- For TIS:

```
void tis_error_handler( file, line, call )
char *file, *call;
int line;
{
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .          your original procedure
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
}
```

- For prober drivers:

```
void prober_error_handler( file, line, call )
char *file, *call;
int line;
{
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .          your original procedure
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
}
```

- For PCL and FCL:

```
void wafer_error_handler( file, line, call )
char *file, *call;
int line;
{
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .          your original procedure
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
}
```

Port Status Function

The Port_status function returns the measurement status of a specified port. This function is useful in getting the saturation or oscillation of an SMU.

Table 4-1 shows all the statuses returned by the Port_status, Rsweep_iv, and Rsweep_miv functions:

Table 4-1. Port Status and its Number

Unit	Status No.	Conditions
SMU	0	Normal.
	1	Another unit reached compliance or limit.
	2	This unit reached compliance or limit.
	3	This unit is oscillating.
	4	Measurement overflow.
CMU	0	Normal.
	6	Measurement value exceeds the set range.
CMU84	0	Normal.
	1	Analog bridge is unbalanced.
	2	A/D converter is not working.
	3	Signal source is overloaded.
	4	ALC[1] is unable to be regulated.

1 ALC is an acronym of Automatic Level Control.

Note

The Port_status function returns the last measurement status after the Sweep_iv or Sweep_miv function is executed.

Reaching the Compliance

You can monitor the saturation status of the SMU using the Port_status function. In other words, the outputs of the SMUs reach the compliance value set by the Force_i, Force_v, Set_iv functions, and so forth.

Figure 4-2 shows an example graph for reaching compliance. The horizontal part of the characteristic curve means that the SMU is reaching the 8 mA current compliance.

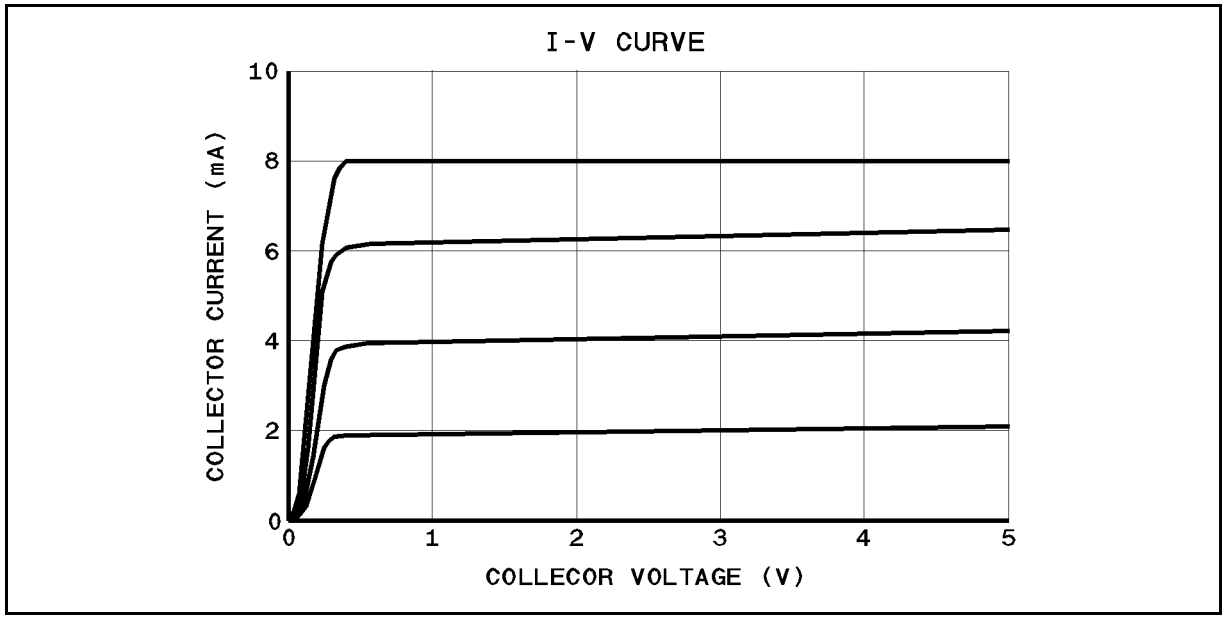


Figure 4-2. Reaching the Compliance (8 mA): Ic-Vce Characteristics

Oscillating

The oscillation of the SMUs can be detected using the Port_status function.

Figure 4-3 shows an example graph for oscillation.

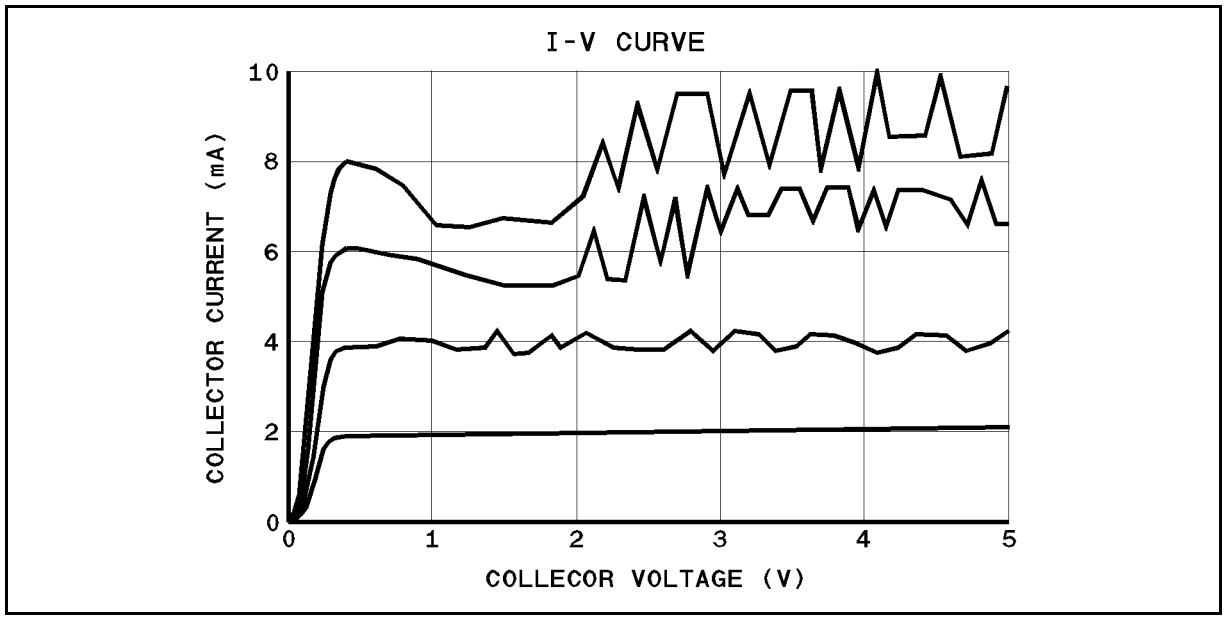


Figure 4-3. Oscillation: Ic-Vce Characteristics

Monitoring with the VFP

This section introduces a method for monitoring the measured data. The X window environment and BASIC/UX are required to perform this method.

The Virtual Front Panel (VFP) program is run in the BASIC/UX window to monitor the measured data of the system instruments, and the C symbolic debugger "`cdb`" can run your test program step by step. Then you can check the measured data in the VFP monitor page.

Briefly, the following procedures are used:

1. Open two windows on the X window environment: one is a terminal and the other is the BASIC/UX window.
2. Load and run the VFP in the BASIC/UX window.
3. Run your test program step by step using `cdb`.

For information on the VFP, see the *System Operation*.

Note

To execute your test program step by step, the test program must be compiled with `-g` option to use the executable file in the `cdb`. For example, type as follows:

```
$ cc -g spot.c -o spot -lhp4062 -lsicl -lcl
```

Then the executable file "`spot`" is created for the `cdb`.

You can monitor the measured data using the VFP even though the test program locks the system instruments using the Lock_system function.

If you monitor the measured data and also change settings, such as the forcing voltage value, the test program must not use the Lock_system function to lock the system instruments. If the source file of the test program includes the Lock_system function, the program lines must be changed to comments using `/*` and `*/` or removed from the test program.

Unlocking the System Instruments

To monitor the measured data from another process, the system instruments must be freed from any invoked processes. To do so, the START program is used for the BASIC/UX environment and `pcsstart` is used for the C environment.

When you run the START program in the BASIC/UX window, "ON-LINE DEBUGGING" must be selected. Then the VFP is loaded and run in the BASIC/UX window.

When you execute the `pcsstart` in the terminal window, the `-on` option must be attached in the command line. For example, type as follows:

```
$ pcsstart -on -v
```

Note that the `pcsstart` does not lock the system instruments. The Lock_system function is used to lock the system instruments in the test program.

Note

START and `pcsstart` are used to initialize the HP 4062UX environment. You only need to execute either START or `pcsstart` for each session. In other words, the START execution in the BASIC/UX window is also available for

executing your test programs developed by the C programming environment in the X window environment.

Monitoring Step by Step with VFP

The VFP is run in the BASIC/UX window and invoked into monitor mode. The VFP periodically updates the measured data of the system instruments. For details about the VFP operation, see the *System Operation*.

You must execute your test program step by step because you must check the measured data displayed in the VFP monitor page. To execute the program step by step, the `cdb` C symbolic debugger is available. The following explains how to execute a program using the `cdb`.

- Getting into the `cdb` environment

  To start the `cdb`, type as follows:

  `$ cdb` *executable_file*

  The *executable_file* is an executable file compiled by `cc` with the `-g` option. If *executable_file* is omitted, `cdb` loads the default executable object file "`a.out`."

- Running a single step

  To run the test program one step at a time, the `s` or `S` command is used as follows:

  ```
  > s
  ```

  To run the program by two or three steps at a time, add a "2" or a "3" prior to the `s` command. For example, as follows:

  ```
  > 2s
  ```

- Setting and deleting a break point

  To set or delete a break point, the `b` or `d` command is used, respectively, as follows:

  `>` *[line]* `b` *setting a break point at the (current) line*
  `> D` *deleting all break points*

- Running or continuing until the next break point

  To run or continue until the next break point, an `r` or `c` is used, respectively, as follows:

  `> r` *running the test program*
  `> c` *continuing until a next break point*

- Displaying the value of a variable

  To display the value of a specified variable, *variable/format* is used as follows:

  `>` *variable/format*

  For example, to display the value of the integer variable "`i`", type as follows:

  ```
  > i/d
  i  =  5
  ```

  The `d` means decimal. For float and double variables, `e` and `f` are used.

- Leaving from `cdb`

To exit from `cdb`, enter the `q` command as follows:

```
> q
Really quit?y
```

For more information on how to use `cdb`, see *HP-UX Concepts and Tutorials: Programming Environment*.

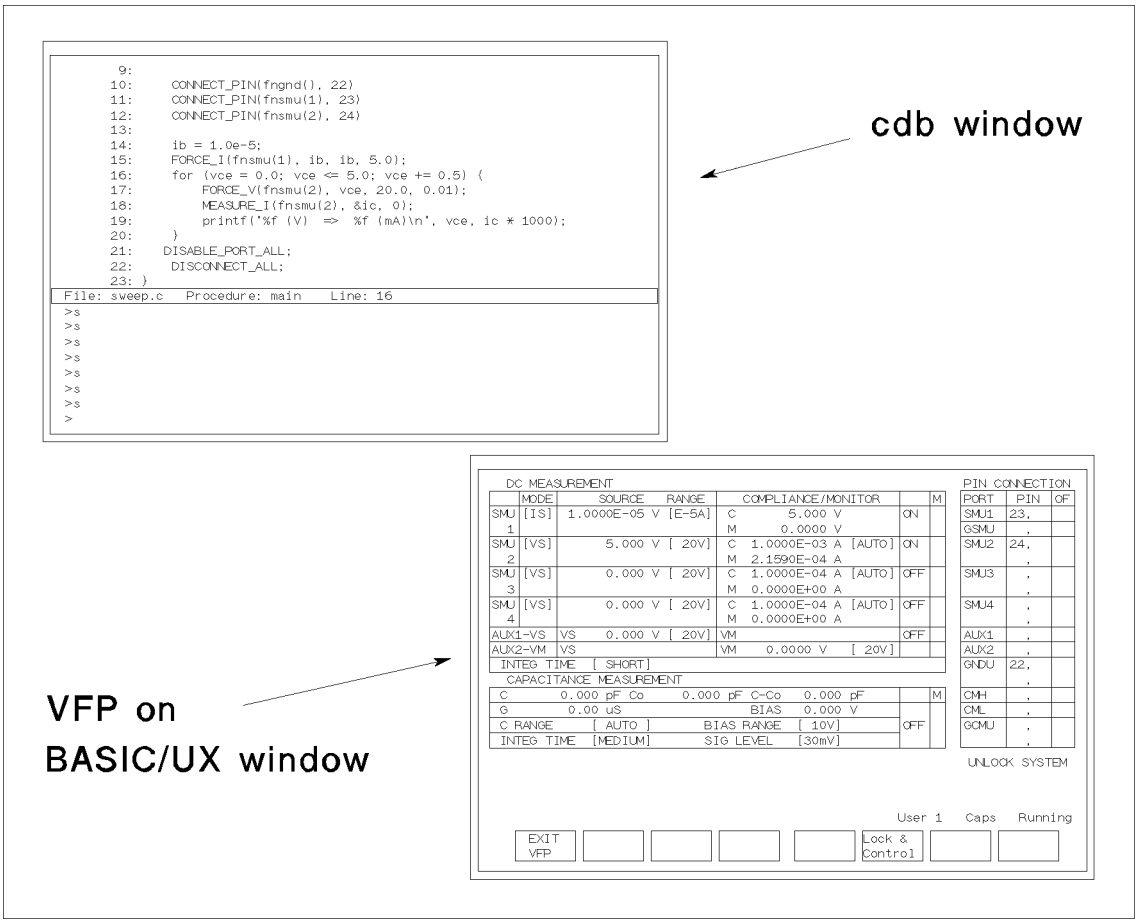


Figure 4-4. Monitoring Step by Step with VFP

Offline Debugging

This section provides information on the offline debugging function.

The offline debugging function gives you the following advantages:

- Maximizes the efficiency of system instrument usage.
- Detects errors early in the test program development stages, without the need to access system instruments.
- Provides a method for developing test programs without an actual test device.

In the offline mode, measured data is simulated automatically or manually and returned to the variables in the test program. There are two simulation modes: constant data simulation mode and file data simulation made. The simulated values are returned to the following functions:

- Force_meas
- Measure_cmu
- Measure_cmu84
- Measure_i
- Measure_p
- Measure_v
- Port_status
- Rsweep_iv
- Rsweep_miv
- Readsweep_dcs
- Readout_dcs
- Search_iv
- Sweep_ct
- Sweep_cv
- Sweep_cv84
- Sweep_iv
- Sweep_miv

Setting to Offline Mode

To get into the offline debugging environment, the START program or `pcsstart` is used. Here, only `pcsstart` is explained. See the *System Operation* for the START program.

The `pcsstart` program requires the `-off` option in the command line for the offline debugging environment as follows:

```
$ pcsstart -off
```

The dummy configuration of the HP 4062UX system depends on the `DUMCONFIG` file in the `/usr/pcs/etc` directory.

Constant Data Simulation Mode

Constant data simulation is the default data simulation mode when you initiate an offline debugging test session (execute the `pcsstart`) or process. You can also set the constant data simulation mode by calling the Off_line_data(NULL) function.

In the constant data simulation mode, TIS functions return constant values, such as range values or compliance values, in accordance with the measurement parameters passed to the TIS functions.

- Force_meas, Measure_i, Measure_v, and Measure_p functions

  Returned values are determined by the value specified as the range parameter. The polarity of the returned value is the same as that of the output value. See the *Programming Reference for C Library* for details on the returned values for each function.

- Rsweep_iv, Rsweep_miv, Sweep_iv, Sweep_miv, Sweep_ct, Sweep_cv, and Sweep_cv84 functions

  These functions return a sequence of linear values. Linear values range from 0 to the specified value of the range parameter. See the *Programming Reference for C Library* for details on the returned values for each function.

- Search_iv function

  A returned search value is the average of the start/minimum and stop/maximum values specified in Set_asearch, Set_bsearch, or Set_lsearch function. See *Programming Reference for C Library* for details on the returned values for each function.

- Measure_cmu and Measure_cmu84 functions

  A returned value is determined by the value specified as the range parameter. See the *Programming Reference for C Library* for details on the returned values for each function.

- Port_status function

  The returned value is always 0 in the constant data simulation mode.

- Readout_dcs and Readsweep_dcs functions

  The returned value for *data* is always 0.1 and the returned value for *status* is always 0.

File Data Simulation Mode

The file data simulation mode is set by calling the Off_line_data function with a specified file name parameter. In this mode, TIS functions get measurement values from the specified file and return them to the proper variables as a simulation values. The file containing the simulation values is called the "offline data simulation file."

The offline data simulation files, which are HP-UX ASCII files, contain a sequence of data records that correspond to their respective TIS functions. Figure 4-5 shows the general syntax of TIS functions and their data records. Parameter items are real numbers.

Data_creation function can help you create an entry data file that you can modify to suit your requirements. See *Programming Reference for C Library* for complete details.

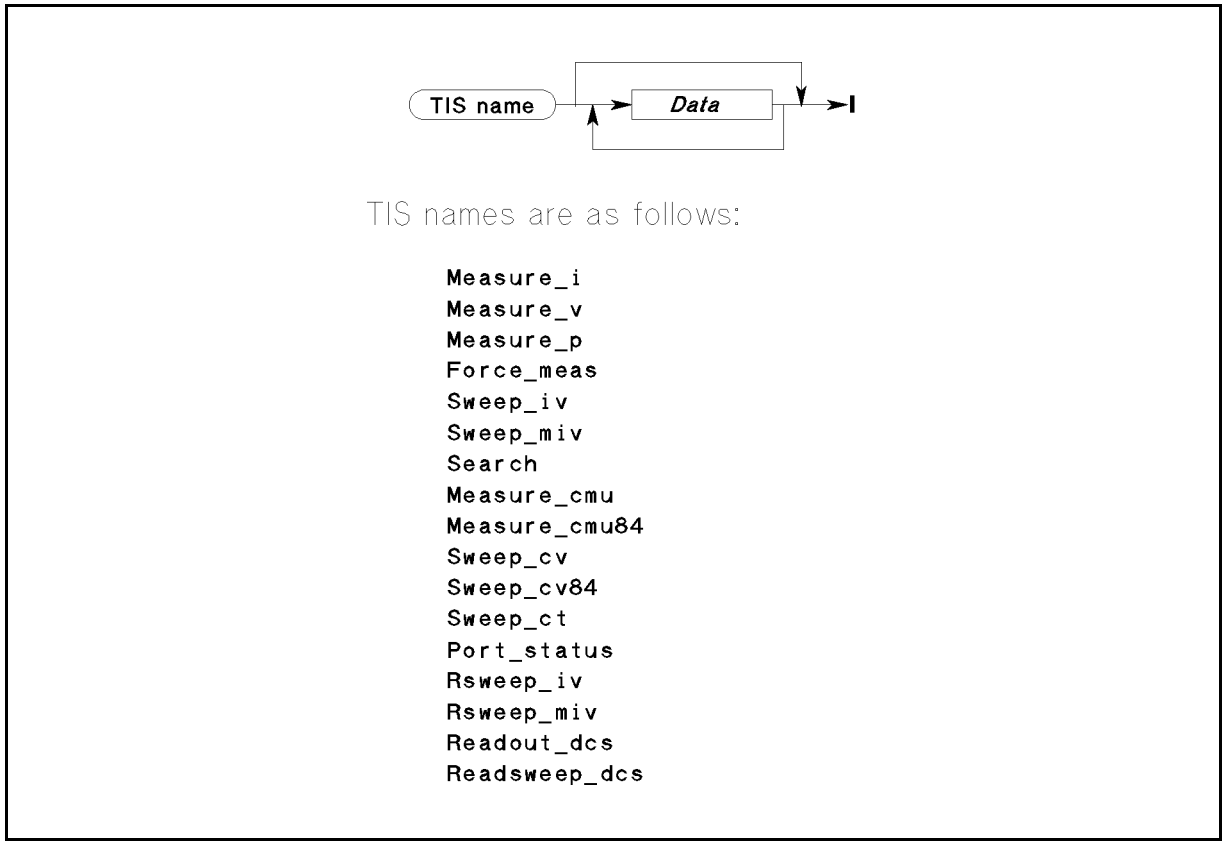


Figure 4-5. Simulation File Data Record Syntax

For TIS sweep functions, one or more array data list representing sweep data is included in the data record. Array data lists must be included within square brackets (`[]`). A right bracket (`[`) indicates the beginning of an array data list, and a left bracket (`]`) indicates the end of an array data list. An array data list consists of a number of data lines, each of which represents a list of real numbers corresponding to parameters of a TIS functions. A white space, tab, or comma can be used to delimit each real number, as in the following example:

```
1000  ! Sweep_iv
1010  ! [
1020  !  1.1355 12.537 4.885E-6 0.0000
1030  !  1.3227 15.831 5.837E-6 0.1008
1040  !  1.4602 16.992 6.135E-6 0.1926
1050  !  1.5062 17.032 6.253E-6 0.3012
1060  ! ]
```

If the number of elements does not fill the corresponding data array, the last element in the list is repeatedly used to fill the rest of the array. Conversely, if the number of elements in the data simulation file exceeds the corresponding array size, extra elements are truncated.

Each line contains an exclamation mark (`!`), indicating a standard BASIC/UX comment line. If a line also ends with an exclamation mark, the offline data simulation function treats that line as a varied program line.

You can edit data records with any text editor, such as `vi`, or the BASIC/UX editor.

Figure 4-6 shows an example of data simulation and its corresponding measurement program.

An example of offline data simulation file name "DATA1":

```
1000  !  Measure_v 1.32197
1010  !  Measure_v 1.85224
1020  !  Sweep_iv
1030  !  [
1040  !  -1.00000E-13, 0.00000E+00
1050  !   3.27391E-03, 0.49999E+00
1060  !   5.01159E-03, 1.00001E+00
1070  !   6.21532E-13, 1050000E+00
1080  !   6.59102E-03, 1.99999E+00
1090  !   6.64221E-03, 2.49999E+00
1100  !   6.64915E-13, 2.99999E+00
1110  !   6.64989E-03, 3.50000E+00
1120  !   6.65023E-03, 4.00001E+00
1130  !   6.65051E-03, 4.50000E+00
1140  !   6.65063E-03, 5.00000E+00
1150  !  ]
```

An programming example:

```
        :
    if (off_line_data("DATA1") == -1) error_rep();

    if (connect_pin2(fnport(2), drain, gate) == -1) error_rep();
    if (connect_pin(fnport(3), substrate) == -1) error_rep();
    if (connect_pin(fngnd, source) == -1) error_rep();

    if (force_v(substrate, v_sub, v_sub, 10.0e-6) == -1) error_rep();
    if (force_i(drain, i_ds1, i_ds2, 2.0) == -1) error_rep();
    if (measure_v(drain, v_th1, 1.8) == -1) error_rep();
    if (force_i(drain, i_ds2, i_ds2, 2.0) == -1) error_rep();
    if (measure_v(drain, v_th2, 1.9) == -1) error_rep();

    if (disable_port3(drain, gate, substrate) == -1) error_rep();
    if (connect_pin(fnport(1), date) == -1) error_rep();

    if (force_v(substrate, v_sub, v_sub, 10.0e-6) == -1) error_rep();
    if (force_v(gate, v_gs, v_gs, 10.0e-6) == -1) error_rep();
    if (set_iv(drain, LINEAR_V, 20.0, 0.0, 5.0, 11, 0.0, 0.0, 10.0e-3, 0.0, COMP_STOP)
        == -1) error_rep();
    if (sweep_iv(drain, I_MEAS, 7.0e-3, i_ds, v_ds, NULL) == -1) error_rep();

    if (disable_port_all() == -1) error_rep();
    if (disconnect_all() == -1) error_rep();

    if (off_line_data(NULL) == -1) error_rep();
        :
```

Figure 4-6. An Example of File Data Simulation

Changing the System Configuration in Offline Mode

To change the system configuration in offline mode, there are two way as follows:

- Modify the "`DUMCONFIG`" file in the `/usr/pcs/etc` directory, then execute `pcsstart` with the "`-off`" option again.
- Create a *DUMCONFIG_file* using the DUMCONFIG format, then execute `pcsstart` with the "`-off` *DUMCONFIG_file*" option again.

Figure 4-7 shows the default configuration of the offline mode. The configuration is for 96-pin SWM system with full pin boards configuration.

SWM Pin Configuration:	1, 2, 3, 4, · · · · · · ·, 94, 95, and 96
DCS Slot Configuration:	Slot 1: HP 41421B 100 V/100 mA SMU Slot 2: — Slot 3: HP 41420A 200 V/1 A SMU Slot 4: HP 41421B 100 V/100 mA SMU Slot 5: HP 41421B 100 V/100 mA SMU Slot 6: HP 41424A VS/VMU Slot 7: — Slot 8: HP 41425A AFU
SMU Port Configuration:	SMU1: 100 V/100 mA SMU SMU2: 200 V/1 A SMU SMU3: 100 V/100 mA SMU SMU4: 100 V/100 mA SMU
AUX Port Configuration:	AUX1: VS1 AUX2: VS2 AUX3: CMH AUX4: CML

Figure 4-7. Default Configuration of Offline Mode

DUMCONFIG Format

The `DUMCONFIG` file in the `/usr/pcs/etc` directory includes the system configuration information used in the offline mode. You can specify the installed pin configuration, the plug-in module type of the DCS, and the SWM port – DCS unit connections in the `DUMCONFIG` file.

You can also specify the system configuration information file for the offline mode using the `-off` *file* option of `pcsstart`. The system configuration information file must be written in the DUMCONFIG format.

The DUMCONFIG can be also used in the BASIC/UX environment, so DUMCONFIG includes the line number and exclamation mark (`!`) at the head of each line as follows:

```
120 ! · · · · · · · · ·
130 ! · · · · · · · · ·
140 ! · · · · · · · · ·
```

You can edit the DUMCONFIG file using the BASIC EDITor or other HP-UX editors, such as the vi editor.

DUMCONFIG includes three kinds of information: the available pin numbers, the plug-in module type of the DCS, and the SWM port – DCS unit connections. The information corresponds to the first, second, and third section of the DUMCONFIG format, respectively. Each section is separated by one or more blank lines. To comment the line, another exclamation mark is used after the line header.

The First section: Available pin numbers

This section includes the available pin numbers. The pin numbers are separated by a comma (`,`), white spaces, tabs, or a new line. If more than 48 pin numbers are specified, the SWM is regarded as a 96-pin SWM system.

The Second section: DCS unit configuration

This section specifies where plug-in modules are installed in the DCS slots. To specify the slot number, the following specifier is used:

`Slot(`*n*`)`

Where *n* shows the slot number: 1 to 8.

One of the following modules must be selected as a plug-in module:

- `HP41420A`
- `HP41421B`
- `HP41424A`
- `HP41425A`

The Third section: SWM port – DCS unit connections

This section specifies the connections between the SWM ports and DCS units. To specify the SWM port, the following specifier is used:

`Port(`*n*`)`	*n = 1 – 9, 21 – 32*[1]
`Smu(`*n*`)`	*n = 1 – 4, 29*[1]*, 32*[1]
`Aux(`*n*`)`	*n = 1 – 4, 11 – 13*[1]*, 21 – 23*[1]*, 31 – 33*[1]*, 41 – 43*[1]
`Cmh`	*is equivalent to* `Aux(3)` *and* `Aux(31)`
`Cml`	*is equivalent to* `Aux(4)` *and* `Aux(41)`

[1] If these number are specified, it is assumed that a port expander is installed on the SWM.

To specify the DCS unit, the following specifier is used:

`Unitsmu(`*n*`)`	$1 \leq n \leq 8$
`Vs(`*n*`)`	$1 \leq n \leq 16$
`Vm(`*n*`)`	$1 \leq n \leq 16$
`Cmu(`*n*`)`	$1 \leq n \leq 2$
`Cmu84(`*n*`)`	$1 \leq n \leq 2$

Figure 4-8 shows an example file of DUMCONFIG. Another exclamation mark after the first one shows a comment line.

```
1000 !!  4062UX offline dummy configuration file
1010 !!  * Switching matrix pin configuration
1020 !     2,  4,  6,  8, 10, 12, 14, 16, 18, 20
1030 !    22, 24, 26, 28, 30, 32, 34, 36, 38, 40
1040 !    42, 44, 46, 48, 50, 52, 54, 56, 58, 60
1050 !    62, 64, 66, 68, 70, 72, 74, 76, 78, 80
1060 !    82, 84, 86, 88, 90, 92, 94, 96
1070 !
1080 !!  * 4142B DC Source/Monitor module configuration
1090 !    Slot(1)     HP41421B
1100 !    Slot(3)     HP41420A
1110 !    Slot(4)     HP41421B
1120 !    Slot(6)     HP41420A
1130 !    Slot(7)     HP41424A
1140 !    Slot(8)     HP41425A
1150 !
1160 !!  * Switching matrix port configuration
1170 !    Smu(1)      Unitsmu(1)
1180 !    Port(2)     Unitsmu(2)
1190 !    Aux(11)     Vs(1)
1200 !    Aux(21)     Vs(2)
1210 !    Aux(12)     Vm(1)
1220 !    Aux(13)     Vm(2)
1230 !    Cmh         Cmu(1)
1240 !    Cml         Cmu(2)
1250 !    Aux(32)     Cmu84(1)
1260 !    Aux(42)     Cmu84(2)
1270 !    Aux(33)     Unitsmu(3)
```

Figure 4-8. An Example File: DUMCONFIG

5

Prober Control

Introduction

This chapter provides the information required to control the following wafer probers using prober drivers specific to each wafer prober:

Electroglas 1034X
Electroglas 2001X
Electroglas 2010X
Electroglas 3001X
Electroglas 4060X
Electroglas 4080X
Tokyo Seimitsu (TSK) A-PM-3000A
Tokyo Seimitsu (TSK) A-PM-6000A
Tokyo Seimitsu (TSK) A-PM-7000A
Tokyo Seimitsu (TSK) A-PM-60A
Tokyo Seimitsu (TSK) A-PM-80A
Tokyo Seimitsu (TSK) A-PM-90A

Included for each wafer prober are instructions on the prober interface settings, the prober control functions, how to compile and execute, a sample program, and prober control commands.

Table 5-1 shows the name of the prober driver functions included in the HP 4062UX system software and a description of each. These functions can be used commonly for Electroglas 1034X/2001X/2010X/3001X/4060X/4080X and TSK A-PM-3000A/A-PM-6000A/A-PM-7000A/A-PM-60A/A-PM-80A/A-PM-90A wafer probers.

Table 5-1. Prober Driver Functions

Function Name	Description
`prober_init()`	Initializes a prober and its prober drivers.
`prober_get_eid()`[1]	Returns the session ID of the HP-IB interface.
`prober_get_ba()`[1]	Returns the bus address of the wafer prober.
`prober_get_name()`[1]	Returns the pointer of the prober model name initialized by `prober_init()`.
`prober_wait()`[1]	Waits for a specified period.
`prober_waittime_ctl()`[1]	Controls the data transfer speed of the HP-IB.
`p_home()`	Unloads the wafer and loading a next wafer.
`p_up()`	Raises the chuck of the wafer prober.
`p_down()`	Lowers the chuck of the wafer prober.
`p_scale()`	Sets the system index for the X- and Y-coordinates.
`p_move()`	Moves the X-Y stage to the specified position absolutely.
`p_imove()`	Moves the X-Y stage to the specified position relatively.
`p_orig()`	Assigns the specified X- and Y-coordinates to the present position of the X-Y stage.
`p_pos()`	Returns the X- and Y-coordinates of the chip currently being probed.
`p_ink()`[2]	Triggers the specified inker of the wafer prober.
`prober_status()`	Returns the current status of the wafer prober.
`prober_reset()`[2]	Sets the wafer prober to Local mode.

1 These functions are used to create new functions for the wafer prober or to create new prober driver functions for the unsupported wafer prober.

2 The `p_ink()` and `prober_reset()` functions are not supported for the Electroglas 1034X wafer prober.

Programming with Prober Drivers

To use the prober driver functions in your test program, the header file "`pcs/prober.h`" must be included in your test program. The `prober.h` header file is stored in the `/usr/include/pcs` directory. To include the `prober.h` header file, use the `#include` preprocessor as follows:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>        including prober.h header file
main()
{
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
}
```

To initialize your wafer prober, the Prober_init function is used. Before any other Prober_*xxxxx* and P_*xxxxx* functions, the Prober_init function must be executed. The first parameter is a pointer to a specifier (strings) that shows the prober model. The prober model and its prober specifier are shown in Table 5-2. The second and third parameters have the logical unit number for the HP-IB and HP-IB address of your wafer prober. If both the second and third parameters are set to 0, the Prober_init function searches the `/usr/pcs/etc/SYSCONFIG` file and finds the logical unit number and HP-IB address of your wafer prober for default settings.

For example, if your system has an Electroglas 2001X wafer prober and the default setting of the logical unit number and HP-IB address of the prober, which is specified in the `/usr/pcs/etc/SYSCONFIG` file, is identified with the "`EG2001X`" entry, you can initialize the prober using the following programming:

```
   .   .   .   .   .   .   .   .
   .   .   .   .   .   .   .   .
if (prober_init("EG2001X", 0, 0) == -1) p_error_rep();
   .   .   .   .   .   .   .   .
   .   .   .   .   .   .   .   .
```

But if the default setting of the logical unit number and HP-IB address of your prober is not identified with the "`EG2001X`" entry, you can initialize the prober specifying the logical unit number of the HP-IB interface and the HP-IB bus address of the prober as follows:

```
   .   .   .   .   .   .   .   .
   .   .   .   .   .   .   .   .
if (prober_init("EG2001X", 25, 4) == -1) p_error_rep();
   .   .   .   .   .   .   .   .
   .   .   .   .   .   .   .   .
```

Table 5-2. Prober Model, Specifier, and Library Name

Prober Model	Specifier	Library File	cc Command Option
Electroglas 1034X	`EGX` or `EG1034X`	`/usr/pcs/lib/libegx.a`	`-legx`
Electroglas 2001X	`EGY` or `EG2001X`	`/usr/pcs/lib/libegy.a`	`-legy`
Electroglas 2010X	`EGY` or `EG2010X`		
Electroglas 3001X	`EGY` or `EG3001X`		
Electroglas 4060X	`EGY` or `EG4060X`		
Electroglas 4080X	`EGY` or `EG4080X`		
TSK A-PM-3000A	`TSK` or `APM3000A`	`/usr/pcs/lib/libtsk.a`	`-ltsk`
TSK A-PM-6000A	`TSK` or `APM6000A`		
TSK A-PM-7000A	`TSK` or `APM7000A`		
TSK A-PM-60A	`TSK` or `APM60A`		
TSK A-PM-80A	`TSK` or `APM80A`		
TSK A-PM-90A	`TSK` or `APM90A`		
Dummy Prober[1]	`DUMMY`	`/usr/pcs/lib/libdmyprob.a`	`-ldmyprob`

1 This prober driver simulates the wafer prober operation without accessing your wafer prober. When you debug a test program that includes a prober controlling operation in offline mode, use the prober type "`DUMMY`" and link this library instead of a prober driver used for online mode such as EGX, which is used for an Electroglas 1034X wafer prober.

Compiling

If your test program includes the instruction for controlling the wafer prober, the prober driver library for your wafer prober must be linked to the executable file as well as the TIS library. The HP 4062UX system supplies four types of prober drivers, which are called the specific prober driver library, as shown in Table 5-2.

To link the specific prober driver libraries using the command option as shown in Table 5-2, another library named `/usr/pcs/lib/libprober.a` is required. The `/usr/pcs/lib/libprober.a`, which is called the generic prober driver library, behaves as an interface between your test program and each prober driver. For example, to compile test program "`sample.c`" and to link two prober driver libraries, use the following command line:

```
$ cc sample.c -lprober -legy -ltsk -lhp4062 -lm -lsicl -lcl
```

Then an executable file "`a.out`" is generated in the current directory.

Note

When the environment variable "`LPATH`" is not set to `/usr/pcs/lib` directory, the following compile error occurs:

```
ld: cannot find library for -lxxxx
ld: (Warning) did not generate an output file
```

Where: *xxxx* is a library name.

For more information on the environment variable "`LPATH`," see "HP-UX Environment Settings" in Chapter 1.

Electroglas 1034X

This section provides information for the Electroglas 1034X wafer prober.

Note

Before using the Electroglas 1034X wafer prober, check the following settings:

HP-IB Address: 1

SRQ Switch: Disabled

Auto or Manual Mode: Auto mode

To control the Electroglas 1034X wafer prober with the prober driver functions, you must set the prober to Auto mode by pressing the [AUTO] key on the Option D interface of the prober. There are no prober control commands using HP-IB for setting the Electroglas 1034X wafer prober to Auto mode.

If the Electroglas 1034X is set to Manual mode while a program is running, program execution pauses when the next prober driver routine executes. To resume program execution, set the Electroglas 1034X to Auto mode.

Preparing to Control the Wafer Prober

Before you can control your Electroglas 1034X using prober driver functions, you must align the probing position by hand. To do so, use the following procedure:

1. Turn on the Electroglas 1034X.
2. Set your Electroglas 1034X to the Manual mode.
3. Align the probing position of the wafer.
4. Raise the chuck using the [Z DRIVE] switch and probe the initial chip.
5. Set your Electroglas 1034X to the Auto mode using [AUTO] switch.

Wafer Prober Control Programming

Setting the System Index and Probing Origin

You must execute the P_scale and P_orig functions before you can move the X-Y stage with either the P_move or P_imove function.

The P_scale function sets the system index for the P_move and P_imove functions initiated X-Y stage movements.

The P_orig function sets the origin for all absolute movements (P_move function) of the prober's X-Y stage by assigning the specified X- and Y-coordinate values to the present probing position, then calculating the origin (0,0) from the system index values specified by the P_scale function.

If the X- and Y-coordinates specified by the P_orig function are (0,0), the present probing position becomes the origin. The present probing position is stored in the prober driver variables. These variables are updated each time the X-Y stage is moved so that the program always knows the present probing position. You must execute the P_orig function after a new wafer is loaded.

Note

Do not use the P_scale and P_orig functions in a prober control program containing a W_start_number or W_start_xy PCL function, because these PCL functions include functions identical to P_scale and P_orig functions. See Chapter 6 for more details.

As an example, the following programming is available:

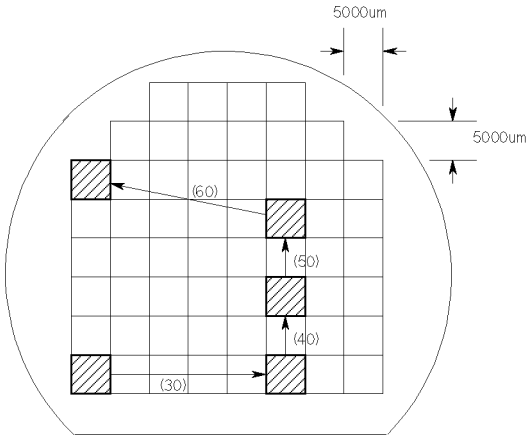
```
#include <stdio.h>
#include <pcs/prober.h>
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
    if (prober_init("EG1034X", 0, 0) == -1) error_rep();
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
    if (p_scale(1000.0, 1000.0) == -1) error_rep();

    printf("Align the first chip position at the probing head\n");
    printf("and set the prober to Auto mode.\n");
    printf("Press [Return] key if you ready.\n");
    while (getchar() != '\n');

    if (p_orig(0.0, 0.0) == -1) error_rep();
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
```

Moving the X-Y Stage

To move the X-Y stage, use either the P_move or P_imove function. The P_move function moves the X-Y stage to the specified X- and Y-coordinates. The P_imove function, on the other hand, moves the X-Y stage to the specified number of system index units. Figure 5-1 shows an example of the probing sequence that results when the programming example is executed.




```
#include <stdio.h>
#include <pcs/prober.h>

main()
{
    if (prober_init("EG1034X", 0, 0) == -1) error_rep();
    if (p_scale(5000.0, 5000.0) == -1) error_rep();
    if (p_orig(0.0, 0.0) == -1) error_rep();
    if (p_move(5.0, 0.0) == -1) error_rep();
    if (p_imove(0.0, 2.0) == -1) error_rep();
    if (p_imove(0.0, 2.0) == -1) error_rep();
    if (p_move(0.0, 5.0) == -1) error_rep();
}
```

Figure 5-1. Moving the X-Y Stage (Electroglas 1034X)

The resolution of X-Y stage movements initiated by the P_move and P_imove function is determined by the system index units set by the P_scale function.

The P_move and P_imove functions initiated movements execute after the prober driver variables containing the X- and Y-coordinates of the present probing position are referenced. These variables are updated upon the completion of a movement. You can recall the values stored in these variables using the P_pos function.

During program execution, the X-Y stage can be moved using the joy stick if the wafer prober is set to Manual mode. The X- and Y-coordinates stored in the prober driver variables before the prober was set to Manual, however, remain unchanged.

Caution

Neither prober drivers nor the wafer prober can detect out-of-range X-Y stage conditions. Be careful when specifying P_move or P_imove function parameters. Improper parameters may cause the wafer forcer to strike the platen frame.

Raising and Lowering the Chuck

Normally, the chuck lowers automatically before the X-Y stage is moved, then raises automatically upon completion of a movement. If the chuck is already down when a movement executes, it remains down at the completion of the movements.

To raise and lower the chuck whenever the X-Y stage is stationary, use the P_up and P_down functions. If the P_up and P_down functions are used in conjunction with any X-Y stage movement (independent from program control), check the probing position using the Prober_status function. If the Prober_status function returns 1 to the second parameter, the probing position is above the wafer.

Loading and Unloading Wafers

To load and unload wafers, use the P_home function. After you execute P_home function and the wafer chuck has returned to its home position, set the VACUUM switch on your Electroglas 1034X wafer prober to off. When P_home function executes, the wafer chuck is lowered, the X-Y stage moves to the home (wafer loading) position, and the prober is set to the Manual mode, at which time you can load or unload a wafer, or perform any manual operations.

Sample Program

The following sample programs show how to control your Electroglas 1034X wafer prober using prober driver functions. Figure 5-2 shows an example program for sequentially probing selected chips, and Figure 5-3 shows an example program for probing all chips in sequential order.

Be sure to compile the example program and link the EGX library before you execute the measurement program. To compile the example program and link the EGX library, use the following `cc` command:

```
$ cc sample.c -lprober -legx -lm -lsicl -lcl
```

In this example, the probing position starts at (0,0), and moves to (4,1), (5,5), (4,6), and (0,0), in that order.

```
#include <stdio.h>
#include <errno.h>
#include <pcs/prober.h>

main()
{
    int auto, onwafer, lastwafer, i;
    double size_x = 5000.0;
    double size_y = 5000.0;
    double x[4] = { 4.0, 5.0, 4.0, 0.0 };  /* Probing position is            */
    double y[4] = { 1.0, 5.0, 6.0, 0.0 };  /* (4,1), (5,5), (4,6), and (0,0) */

    if (prober_init("EG1034X", 0, 0) == -1) p_error_rep();

    if (p_scale(size_x, size_y) == -1) p_error_rep();
    printf("Align the wafer and press [AUTO] key.\n");
    do
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
    while (auto == 0);

    if (p_orig(0.0, 0.0) == -1) p_error_rep();

    for (i = 0; i < 4; i++) {
        if (p_move(x[i], y[i]) == -1) p_error_rep();
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
        if (onwafer) test();
    }
    if (p_home() == -1) p_error_rep();
}
test()
{
    printf("Measuring\n");
}
p_error_rep()
{
    fprintf(stderr, "p_errno = %d, errno = %d.\n", p_errno, errno);
    exit(-1);
}
```

Figure 5-2. Sample Program: Probing Selected Chips (Electroglas 1034X)

In this example, the probing stage starts at (0,0), and moves to the next position until it comes to the last position. If there is a chip at the probing position, the measurement routine is executed.

```
#include <stdio.h>
#include <errno.h>
#include <pcs/prober.h>

main()
{
    int auto, onwafer, lastwafer, i, j;
    double size_x = 5000.0;
    double size_y = 5000.0;
    double chip_x = 7.0;
    double chip_y = 7.0;
    double x, y;

    if (prober_init("EG1034X", 0, 0) == -1) p_error_rep();

    if (p_scale(size_x, size_y) == -1) p_error_rep();
    printf("Align the wafer and press [AUTO] key.\n");
    do
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
    while (auto == 0);

    if (p_orig(0.0, 0.0) == -1) p_error_rep();

    for (y = 0.0; y <= chip_y; y += 1.0)
        for (x = 0.0; x <= chip_x; x += 1.0) {
            if (p_move(x, y) == -1) p_error_rep();
            if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
            if (onwafer) test();
        }
    }
    if (p_home() == -1) p_error_rep();
}
test()
{
    printf("Measuring\n");
}
p_error_rep()
{
    fprintf(stderr, "p_errno = %d, errno = %d.\n", p_errno, errno);
    exit(-1);
}
```

Figure 5-3. Sample Program: Probing all Chips (Electroglas 1034X)

Prober Control Commands

The prober control commands used to control the Electroglas 1034X in the prober driver functions are shown in Table 5-3.

Table 5-3. Prober Control Commands (Electroglas 1034X)

Function	Prober Control Command	Returned Code
Move chuck to the specified X-Y coordinates	MOX±*xxxxx*Y±*xxxxx*[1]	MC
Chuck up	ZU	MC
Chuck down	ZD	MC
Chuck home	ZH	MC
Set the current position at the origin	PRX±*xxxxx*Y±*xxxxx*[1]	
Check prober status	?S	SZ*ccbcb*[2]

1 *xxxxx* represents any integer number.

2 *c* represents any single character and *b* represents either 1 or 0.

Electroglas 2001X/2010X/3001X/4060X/4080X

This section provides information for the Electroglas 2001X/2010X/3001X/4060X/4080X wafer prober.

Note

Before using the Electroglas 2001X wafer prober, check the following settings:

Set mode display	METRIC/ENGLISH: METRIC AUTO PROBER PATTERN: EXTERNAL I/O PROTOCOL: STANDARD
External I/O parameters	I/O PORT: GP-IB GP-IB ADDRESS: 2
Set option display	AUTO LOAD: ENB AUTO ALIGN: ENB AUTO PROFILE: ENB
SRQ Switch (A4 SW1-8):	ENB (open)

For the settings of the Electroglas 2010X/3001X/4060X/4080X, see the prober manuals.

Note

If your Electroglas 2001X/2010X/3001X/4060X/4080X is not equipped with an auto loader, a profiler, or an auto aligner, the prober driver library routine must be modified. For details, see Appendix A.

Preparing to Control the Wafer Prober

Before you can control your Electroglas 2001X/2010X/3001X/4060X/4080X using prober driver functions, you must set your prober to the *Remote* mode. There are no HP-IB commands for setting your Electroglas 2001X/2010X/3001X/4060X/4080X to Remote. Your prober's RUN TIME display must display "IDLE" and "ON LINE" before you execute your program.

To control your Electroglas 2001X/2010X/3001X/4060X/4080X using prober driver functions, perform the following procedure:

1. Turn on the Electroglas 2001X/2010X/3001X/4060X/4080X.
2. Press the cassette switches.
3. Press the switch on the left side of the joystick.
4. Set your prober as described in the note above.
5. Load, align, and profile the wafer as follows:
   a. Press [LOAD] on the joystick unit.
   b. Press the [FID TARG] key on the Monitor Unit.
   c. Use the joystick to set the probing position.
   d. Press the [PAUSE] key.
   e. Press the [ALIGN SCAN] key.
   f. Use the joystick to move the wafer to the first area.

g. Press the [FIRST] key.

6. Press the [AUTOPROBE] key on the operator console to lock out the keyboard (sets your Electroglas 2001X/2010X/3001X/4060X/4080X to Remote). The run time display changes from "IDLE" to "PROBE".

Pressing the [AUTOPROBE] key locks out the keyboard of your Electroglas 2001X/2010X/3001X/4060X/4080X and sends a status message telling the controller that prober setup is complete and testing may begin. To determine whether your prober is set to Remote or Local, use the Prober_status function.

Note

To terminate the program execution, send the INTR signal to the executed process. To do so, press [Break] on the keyboard for the default terminal configuration. For details on the technical configuration, see the STTY(1) reference page of the *HP-UX reference* manual.

Wafer Prober Control Programming

Setting the System Index and Probing Origin

You must execute the P_scale and P_orig functions before you can move the X-Y stage using either the P_move or P_imove function.

The P_scale function sets the system index for the P_move and P_imove functions initiated X-Y stage movements. You can execute the P_scale function before you set your wafer prober to the Remote mode.

The P_orig function sets the origin for all absolute movements (P_move function) of the prober's X-Y stage by assigning the specified X- and Y-coordinate values to the present probing position, then calculating the origin (0,0) from the system index values specified by the P_scale function.

If the X- and Y-coordinates specified by the P_orig function are (0,0), the present probing position becomes the origin. The present probing position is stored in the prober driver variables. These variables are updated each time the X-Y stage is moved so that the program always knows the present probing position. You must execute the P_orig function after a new wafer is loaded.

Note

Do not include the P_scale and P_orig functions in a prober control program that contains a W_start_number or W_start_xy PCL function, because these PCL functions include functions identical to P_scale and P_orig function. Refer to Chapter 6 for details.

As an example, the following programming is available:

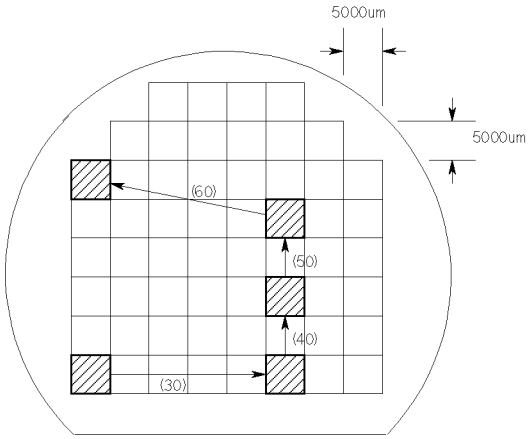
```
#include <stdio.h>
#include <pcs/prober.h>
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
    if (prober_init("EG2001X", 0, 0) == -1) error_rep();
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
    if (p_scale(1000.0, 1000.0) == -1) error_rep();

    printf("Align the first chip position at the probing head\n");
    printf("and set the prober to Auto mode.\n");
    printf("Press [Return] key if you ready.\n");
    while (getchar() != '\n');

    if (p_orig(0.0, 0.0) == -1) error_rep();
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
```

Moving the X-Y Stage

To move the X-Y stage, use either the P_move or P_imove function. The P_move function moves the X-Y stage to the specified X- and Y-coordinates. The P_imove function, on the other hand, moves the X-Y stage to the specified number of system index units. Figure 5-4 shows the probing sequence that results when the programming example is executed.




```
#include <stdio.h>
#include <pcs/prober.h>

main()
{
    if (prober_init("EG2001X", 0, 0) == -1) error_rep();
    if (p_scale(5000.0, 5000.0) == -1) error_rep();
    if (p_orig(0.0, 0.0) == -1) error_rep();
    if (p_move(5.0, 0.0) == -1) error_rep();
    if (p_imove(0.0, 2.0) == -1) error_rep();
    if (p_imove(0.0, 2.0) == -1) error_rep();
    if (p_move(0.0, 5.0) == -1) error_rep();
}
```

Figure 5-4. Moving the X-Y Stage (Electroglas 2001X)

The resolution of X-Y stage movements initiated by the P_move and P_imove functions is determined by the system index units set by the P_scale function.

The P_move and P_imove functions initiated movements execute after the prober driver variables containing the X- and Y-coordinates of the present probing position are referenced. These variables are updated upon the completion of a movement. You can recall the values stored in these variables using the P_pos function.

During program execution, you can move the X-Y stage using the joystick if the prober is set to PAUSE or IDLE mode. The X- and Y-coordinates stored in the prober driver variables before the prober was set to PAUSE or IDLE, however, remain unchanged.

Note

Operator console initiated X-Y stage movements are in the index units set from the operator console.

Raising and Lowering the Chuck

Normally, the chuck lowers automatically before the X-Y stage moves. To raise the chuck when a movement completes, use the P_up function. If the chuck is already down when a movement executes, it remains down at the completion of the movement. You can execute the P_up and P_down functions when the X-Y stage is stationary.

Inker Trigger

To trigger your prober's inker function, use the P_ink function and specify an inker bin code as the parameter. The actual inkers to be triggered by the inker bin code depend on the preset inker assignment.

Loading and Unloading Wafers

To load and unload wafers, use the P_home function. If the wafer prober is ON LINE, P_home function performs the following operations:

1. Unloads the wafer from the X-Y stage.
2. Loads, profiles, and aligns the next wafer.
3. Moves the X-Y stage to the initial probing position.

During this sequence of operations, the wafer prober remains ON LINE. After the X-Y stage is positioned at the initial probing position, the wafer prober is automatically set to the Remote mode.

Before executing any other prober control functions, the program must check the prober status to confirm that the prober is Remote. When the last wafer is unloaded, the prober is not returned to Remote, and this causes prober driver error 21. Check this flag and Remote mode status by using the Prober_status function.

Checking Prober Status

To check your prober's Remote or Local status, Prober_status function can be used. The synopsis of the Prober_status function is as follows:

```
int prober_status(remote, on_wafer, last_wafer)
int *remote, *on_wafer, *last_wafer;
```

If the value returned to *remote* is 1, the wafer prober is in the Remote mode. *On_wafer* indicates whether the present probing position is on the wafer or not, and is altered when the

P_move, P_imove, or P_up function executes. A 1 is returned to *on_wafer* when the probe is on the wafer. *Last_wafer* is a dummy parameter.

Sample Program

The following sample programs show how to control your Electroglas 2001X wafer prober using prober driver functions. Figure 5-5 shows an example program for sequentially probing selected chips, and Figure 5-6 shows an example program for probing all chips in sequential order.

Be sure to compile the example program and link the EGY library before you execute the measurement program. To compile the example program and link the EGY library, use the following `cc` command:

```
$ cc sample.c -lprober -legy -lm -lsicl -lcl
```

In this example, the probing position starts at (0,0), and moves to (4,1), (5,5), (4,6), and (0,0), in that order.

```
#include <stdio.h>
#include <errno.h>
#include <pcs/prober.h>

main()
{
    int auto, onwafer, lastwafer, i;
    double size_x = 5000.0;
    double size_y = 5000.0;
    double x[4] = { 4.0, 5.0, 4.0, 0.0 };  /* Probing position is           */
    double y[4] = { 1.0, 5.0, 6.0, 0.0 };  /* (4,1), (5,5), (4,6), and (0,0) */

    if (prober_init("EG2001X", 0, 0) == -1) p_error_rep();

    if (p_scale(size_x, size_y) == -1) p_error_rep();
    printf("Align the wafer and press [AUTO PROBE].\n");
    do
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
    while (auto == 0);

    if (p_orig(0.0, 0.0) == -1) p_error_rep();

    for (i = 0; i < 4; i++) {
        if (p_move(x[i], y[i]) == -1) p_error_rep();
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
        if (onwafer) test();
    }
    if (p_home() == -1) p_error_rep();
}
test()
{
    printf("Measuring\n");
}
p_error_rep()
{
    fprintf(stderr, "p_errno = %d, errno = %d.\n", p_errno, errno);
    exit(-1);
}
```

Figure 5-5.
Sample Program: Probing Selected Chips (Electroglas 2001X/2010X/3001X/4060X/4080X)

In this example, the probing stage starts at (0,0), and moves to the next position until it comes to the last position. If there is a chip at the probing position, the measurement routine is executed.

```
#include <stdio.h>
#include <errno.h>
#include <pcs/prober.h>

main()
{
    int auto, onwafer, lastwafer, i, j;
    double size_x = 5000.0;
    double size_y = 5000.0;
    double chip_x = 7.0;
    double chip_y = 7.0;
    double x, y;

    if (prober_init("EG2001X", 0, 0) == -1) p_error_rep();

    if (p_scale(size_x, size_y) == -1) p_error_rep();
    printf("Align the wafer and press [AUTO PROBE].\n");
    do
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
    while (auto == 0);

    if (p_orig(0.0, 0.0) == -1) p_error_rep();

    for (y = 0.0; y <= chip_y; y += 1.0)
        for (x = 0.0; x <= chip_x; x += 1.0) {
            if (p_move(x, y) == -1) p_error_rep();
            if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
            if (onwafer) test();
        }
    }
    if (p_home() == -1) p_error_rep();
}
test()
{
    printf("Measuring\n");
}
p_error_rep()
{
    fprintf(stderr, "p_errno = %d, errno = %d.\n", p_errno, errno);
    exit(-1);
}
```

Figure 5-6.
Sample Program: Probing all Chips (Electroglas 2001X/2010X/3001X/4060X/4080X)

Prober Control Commands

The prober control commands used to control the Electroglas 2001X in the prober driver functions are shown in Table 5-4.

Table 5-4.
Prober Control Commands (Electroglas 2001X/2010X/3001X/4060X/4080X)

Function	Prober Control Command	Returned Code[1]
Move chuck to the specified X-Y coordinates	MMX±*xxxxx*Y±*xxxxx*[2]	MC[3]
Chuck up	ZU	MC[3]
Chuck down	ZD	
Chuck home	LO	MC[3]
Auto profile	PZ	
Auto Align	AA	MC[3]
Trigger an inker	IK*c*[4]	
Check prober status	?S	SZ*c*W*b*C*b*[5]

1 The Electroglas 2001X/2010X/3001X/4060X/4080X must be set to a true SRQ line before outputting data.

2 *xxxxx* represents any integer number.

3 MF is returned if the command response fails.

4 *c* represents any ASCII characters between "0" – "9", ":", ";", "<", "=" , ">" ,and "?".

5 *c* represents either U or D, and *b* represents either 1 or 0.

TSK A-PM-3000A/6000A/7000A/60A/80A/90A

This section provides information to control the Tokyo Seimitsu (TSK) A-PM-3000A/6000A/7000A/60A/80A/90A wafer probers using the prober driver functions.

Note

Before using the TSK A-PM-3000A/6000A/7000A/60A/80A/90A wafer prober, you must make the proper settings. For the TSK A-PM-3000A/6000A/7000A, use the following settings:

GP-IB Address:	5
Mode Switches	3-4: OFF 3-5: ON 23-4 ON

For the TSK A-PM-60A/80A/90A wafer prober, see the prober manuals.

Preparing to Control the Wafer Prober

Before you can control your TSK prober using the prober driver functions, you must set your prober to the Remote mode. To set your TSK prober to Remote, press the DATA IN switch after you align a wafer in GP-IB mode. Use the Prober_status function to confirm the prober's Remote status. TSK probers are set to Local when a P_home or Prober_reset function is executed.

Wafer Prober Control Programming

Setting the System Index and Probing Origin

You must execute the P_scale and P_orig functions before you can move the X-Y stage using either the P_move or P_imove function.

The P_scale function sets the system index for the P_move and P_imove functions initiated X-Y stage movements. You can execute the P_scale function before you set your wafer prober to the Remote mode.

The P_orig function sets the origin for all absolute movements (P_move function) of the prober's X-Y stage by assigning the specified X- and Y-coordinate values to the present probing position, then calculating the origin (0,0) from the system index values specified by the P_scale function.

If the X- and Y-coordinates specified by P_orig function are (0,0), the present probing position becomes the origin. The present probing position is stored in the prober driver variables. These variables are updated each time the X-Y stage is moved so that the program always knows the present probing position. You must execute P_orig function after a new wafer is loaded.

Note

Do not include the P_scale and P_orig functions in a prober control program that contains a W_start_number or W_start_xy PCL function, because these PCL functions include functions identical to the P_scale and P_orig functions. Refer to Chapter 6 for details.

As an example, the following programming is available:

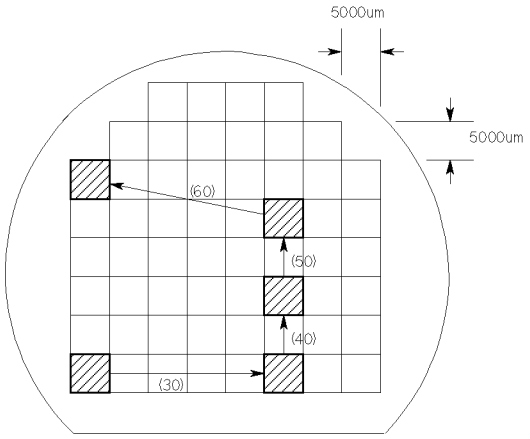
```
#include <stdio.h>
#include <pcs/prober.h>
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
    if (prober_init("APM7000A", 0, 0) == -1) error_rep();
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
    if (p_scale(1000.0, 1000.0) == -1) error_rep();

    printf("Align the first chip position at the probing head\n");
    printf("and press [DATA IN] key.\n");
    printf("Press [Return] key if you ready.\n");
    while (getchar() != '\n');

    if (p_orig(0.0, 0.0) == -1) error_rep();
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
```

Moving the X-Y Stage

To move the X-Y stage, use either the P_move or P_imove function. The P_move function moves the X-Y stage to the specified X- and Y-coordinates. The P_imove function, on the other hand, moves the X-Y stage to the specified number of system index units. Figure 5-7 shows the probing sequence that results when the programming example is executed.




```
#include <stdio.h>
#include <pcs/prober.h>

main()
{
    if (prober_init('APM7000A', 0, 0) == -1) error_rep();
    if (p_scale(5000.0, 5000.0) == -1) error_rep();
    if (p_orig(0.0, 0.0) == -1) error_rep();
    if (p_move(5.0, 0.0) == -1) error_rep();
    if (p_imove(0.0, 2.0) == -1) error_rep();
    if (p_imove(0.0, 2.0) == -1) error_rep();
    if (p_move(0.0, 5.0) == -1) error_rep();
}
```

Figure 5-7. Moving the X-Y Stage (TSK A-PM-3000A/6000A/7000A/60A/80A)

The resolution of X-Y stage movements initiated by the P_move and P_imove function is determined by the system index units set by the P_scale function.

The P_move and P_imove functions initiated movements execute after the prober driver variables containing the X- and Y-coordinates of the present probing position are referenced. These variables are updated upon the completion of a movement. You can recall the values stored in these variables using the P_pos function.

The P_move and P_imove functions halt your TSK prober's processor to minimize measurement errors caused by system noise. Therefore, the prober's panel controls are disabled.

Raising and Lowering the Chuck

Normally, the chuck lowers automatically before the X-Y stage moves, then raises automatically at the completion of a movement. If the chuck is already down when a movement executes, it remains down at the completion of a movement. You can execute P_up and P_down functions at any time when the X-Y stage is stationary.

The P_up function halts your TSK prober's processor after the chuck is raised to minimize measurement errors caused by noise. P_down function releases the processor from its halt state.

Inker Trigger

To trigger your prober's inker function, use the P_ink function and specify an inker number between 1 and 4 as a parameter. If the processor is halted before P_ink function executes, the processor is released from its halt condition.

Loading and Unloading Wafers

To load and unload wafers, use the P_home function. If the wafer prober's UNLOAD switch is set to AUTO, the P_home function performs the following operations:

1. Unloads the wafer from the X-Y stage.
2. Loads and aligns the next wafer.
3. Moves the X-Y stage to the initial probing position.

During this sequence of operations, the wafer prober is released from the Remote mode. After the X-Y stage is positioned at the initial probing position, the wafer prober is automatically set to the Remote mode.

Before executing any other prober control functions, the program must check the prober status to confirm that the prober is Remote. When the last wafer is unloaded, the prober is not returned to Remote, but sets a flag indicating there are no more wafers in the wafer stack. Check this flag and Remote mode status by using the Prober_status function.

Checking Prober Status

To check your prober's Remote or Local status, the Prober_status function can be used. The synopsis of the Prober_status function is as follows:

```
int prober_status(remote, on_wafer, last_wafer)
int *remote, *on_wafer, *last_wafer;
```

If the value returned to *remote* is 1, the wafer prober is in the Remote mode.

On_wafer indicates whether the present probing position is on the wafer or not, and is altered when P_move, P_imove, or P_up function executes. A 1 is returned to *on_wafer* when the probe is on the wafer.

Last_wafer indicates wafer stack status. After the last wafer is unloaded from the chuck by the P_home function, this variable is set to 1 and the prober is set to the Local mode. *Last_wafer* is always 0 if your prober is *not* equipped with an automatic wafer loader.

Sample Program

The following sample programs show how to control your TSK A-PM-7000A wafer prober using prober driver functions. Figure 5-8 shows an example program for sequentially probing selected chips, and Figure 5-9 shows an example program for probing all chips in sequential order.

Be sure to compile the example program and link the TSK library before you execute the measurement program. To compile the example program and link the TSK library, use the following `cc` command:

```
$ cc sample.c -lprober -ltsk -lm -lsicl -lcl
```

In this example, the probing position starts at (0,0), and moves to (4,1), (5,5), (4,6), and (0,0), in that order.

```
#include <stdio.h>
#include <errno.h>
#include <pcs/prober.h>

main()
{
    int auto, onwafer, lastwafer, i;
    double size_x = 5000.0;
    double size_y = 5000.0;
    double x[4] = { 4.0, 5.0, 4.0, 0.0 };  /* Probing position is           */
    double y[4] = { 1.0, 5.0, 6.0, 0.0 };  /* (4,1), (5,5), (4,6), and (0,0) */

    if (prober_init("APM7000A", 0, 0) == -1) p_error_rep();

    if (p_scale(size_x, size_y) == -1) p_error_rep();
    printf("Align the wafer and press [DATA IN].\n");
    do
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
    while (auto == 0);

    if (p_orig(0.0, 0.0) == -1) p_error_rep();

    for (i = 0; i < 4; i++) {
        if (p_move(x[i], y[i]) == -1) p_error_rep();
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
        if (onwafer) test();
    }
    if (p_home() == -1) p_error_rep();
}
test()
{
    printf("Measuring\n");
}
p_error_rep()
{
    fprintf(stderr, "p_errno = %d, errno = %d.\n", p_errno, errno);
    exit(-1);
}
```

Figure 5-8. Sample Program: Probing Selected Chips (TSK Probers)

In this example, the probing stage starts at (0,0), and moves to the next position until it comes to the last position. If there is a chip at the probing position, the measurement routine is executed.

```
#include <stdio.h>
#include <errno.h>
#include <pcs/prober.h>

main()
{
    int auto, onwafer, lastwafer, i, j;
    double size_x = 5000.0;
    double size_y = 5000.0;
    double chip_x = 7.0;
    double chip_y = 7.0;
    double x, y;

    if (prober_init("APM7000A", 0, 0) == -1) p_error_rep();

    if (p_scale(size_x, size_y) == -1) p_error_rep();
    printf("Align the wafer and press [DATA IN].\n");
    do
        if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
    while (auto == 0);

    if (p_orig(0, 0) == -1) p_error_rep();

    for (y = 0.0; y <= chip_y; y += 1.0)
        for (x = 0.0; x <= chip_x; x += 1.0) {
            if (p_move(x, y) == -1) p_error_rep();
            if (prober_status(&auto, &onwafer, &lastwafer) == -1) p_error_rep();
            if (onwafer) test();
        }
    }
    if (p_home() == -1) p_error_rep();
}
test()
{
    printf("Measuring\n");
}
p_error_rep()
{
    fprintf(stderr, "p_errno = %d, errno = %d.\n", p_errno, errno);
    exit(-1);
}
```

Figure 5-9. Sample Program: Probing all Chips (TSK Probers)

Prober Control Commands

The prober control commands used to control the TSK probers in the prober driver functions are shown in Table 5-5.

Table 5-5. Prober Control Commands (TSK Probers)

Function	Prober Control Command	Returned Code
Move chuck to the specified X-Y coordinates	AY±*xxxxx*X±*xxxxx*[1]	65
Chuck up	Z	67 or 73[2]
Chuck down	D	68
CPU halt	T	
Trigger an inker	M*n*[3]	
Chuck home	L	70 or 76[4]

1 *xxxxx* represents any integer number.

2 The 73 indicates Edge sensor is ON.

3 *n* represents any ASCII characters between "1" – "4", which indicates an inker number.

4 The 76 (error SRQ) indicates cassette in empty. If the 76 is returned, send "E0010" string to the prober.

6

Wafer Probing and Data Logging

Introduction

This chapter describes the system's wafer probing and data logging software and provides information on how to effectively use this software for performing wafer measurements with an automatic wafer prober.

The software described in this chapter consists of two libraries as follows:

Probing Control Library (PCL)	Is a set of functions for controlling a wafer prober in accordance with probing pattern data stored in a probing pattern data file.
File Creation Library (FCL)	Is a set of functions that convert measurement results into the format required for wafer mapping and statistical graphics, and stores this data in an FCL-created measurement data file.

The PCL and FCL are stored in the following files:

Header file:	`/usr/include/pcs/wafer.h`
Object file:	`/usr/pcs/lib/libhp4062.a`

Source files:	`/usr/pcs/src/wafer/wafer_com.c`
	`/usr/pcs/src/wafer/wafer_fcl.c`
	`/usr/pcs/src/wafer/wafer_pcl.c`
	`/usr/pcs/src/wafer/waferrh.c`

Wafer Measurement Software Requirements

Figure 6-1 provides an example of the organization and data flow of a typical wafer measurement. The figure also shows the software required during each measurement phase.

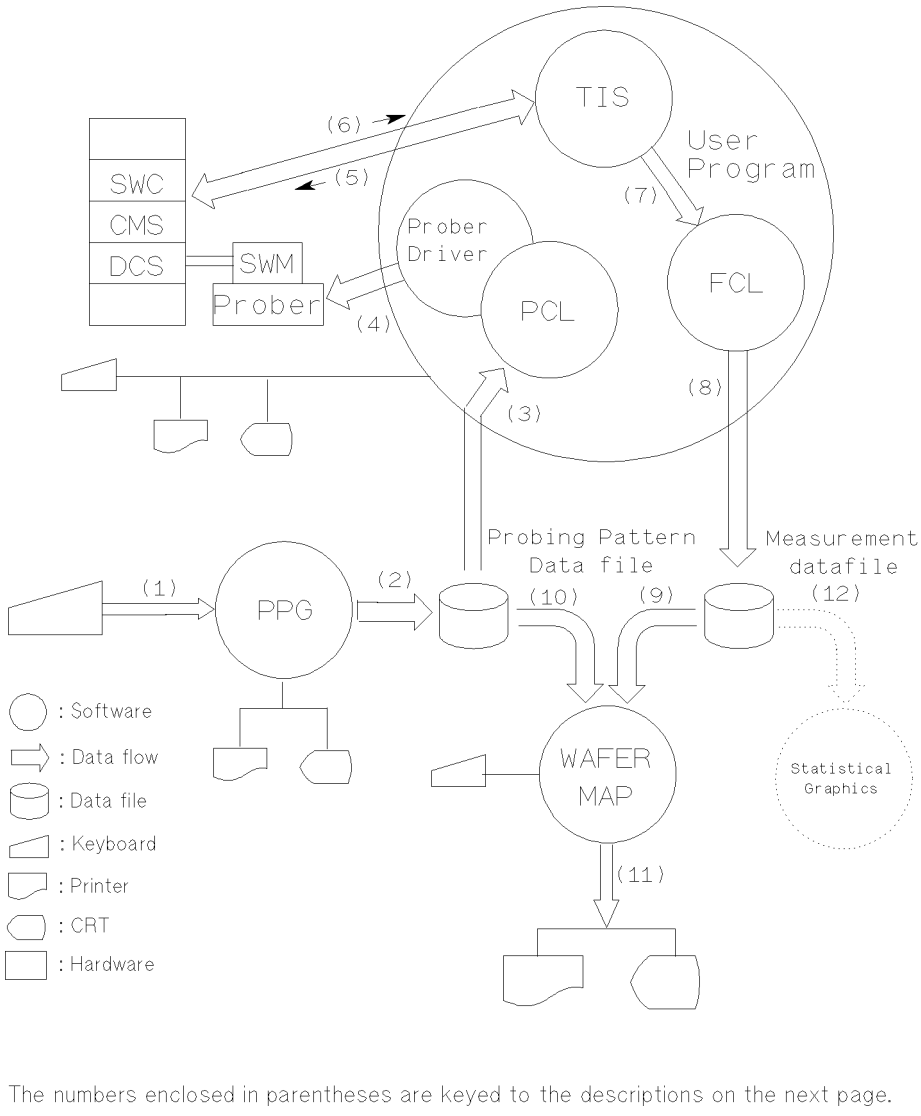


Figure 6-1. Wafer Measurement Data Flow and Software Requirements

- Probing Pattern Generator (PPG)

  (1) PPG prompts you to enter the data necessary to create a probing pattern.

  (2) PPG creates the *probing pattern data file*, and stores the data you input into the file. This data file can be updated whenever necessary.

- Probing Control Library (PCL)

(3) The PCL function loads the probing pattern data file created in (2).

(4) PCL controls the wafer prober in accordance with the probing pattern data.

- TIS, user libraries, etc.

(5) You can use functions, such as TIS and user libraries, to facilitate measurement programming.

(6) Wafer measurement results are returned to TIS and user libraries.

- File Creation Library (FCL)

(7) You can use the FCL functions to convert the measurement results into the format required for wafer mapping and statistical graphics.

(8) FCL creates a measurement data file, and stores the data from (7) into the file.

- WAFER MAP

(9) WAFER MAP loads the measurement data file generated in (8).

(10) WAFER MAP loads the probing pattern data file generated in (2).

(11) WAFER MAP uses the measurement data and probing pattern data to create a wafer map, then displays the map on the CRT or outputs the map to a printer.

Note that the PPG and WAFER MAP are HP 4062 utility software executed on the BASIC/UX environment. For the descriptions on these software, see Chapter 6 in the HP 4062C/UX *System Operation* manual.

Compiling and Executing

This section shows how to compile a test programs that contains PCL or FCL function calls, and how to execute the test program.

The PCL and FCL functions are provided by the `/usr/pcs/lib/libhp4062.a` library file. Moreover, the PCL functions use the generic prober driver library routines in the `libprober.a` and prober model specific library file such as `libtsk.a` and `libegx.a` in the `/usr/pcs/lib` directory. When compiling a program that includes PCL functions, you must specify the necessary library file names in the `cc` command line. For the prober drivers, see Chapter 5.

For example, when your program `probing.c` includes the routines to control the Electroglas 1034X wafer prober only, start the C compiler as follows:

```
$ cc probing.c -lprober -legx -lhp4062 -lm -lsicl -lcl
```

You can also link more than one type of prober driver. For example, to link the Electroglas 1034X and 2001X wafer probers, execute the C compiler as follows:

```
$ cc probing.c -lprober -legx -legy -lhp4062 -lm -lsicl -lcl
```

The C compiler creates an executive file `a.out`.

Note

When linking library files by using the `cc` command, be aware of the following rules:

- `-lprober` must be specified before linkage specifications (`-l` options) for any specific prober libraries such as `-legx`.
- No prober specific library linkage specifications can appear after `-lhp4062`.
- There are no restrictions to specify the prober specific library linkage specifications.

Note

Be sure to set the wafer prober to Remote and to manually probe the first chip before actually attempting to perform any measurements. Refer to Chapter 5.

For example, type the following to execute your test program "`probing.c`" after executing `pcsstart`:

```
$ a.out
```

If your program controls your wafer prober and not the system instruments, you do not need to execute `pcsstart` previously. Because `pcsstart` initialize the system instruments only and does not access the wafer prober.

Tip

The default output file name from the linker in the `cc` procedure is "`a.out`". You can specify the output file name in the `cc` command line using `-o` *name* option. For example, to compile the file `source.c` and create the executable file `source`, type the following:

```
$ cc source.c -o source -lprober -legx -lhp4062 -lm -lsicl -lcl
```

Note

If the environment variable LPATH is not set to "`/usr/pcs/lib`," the `cc` procedure reports errors, for example, as follows:

```
ld: cannot find library for -lprober
ld: cannot find library for -legx
ld: cannot find library for -lhp4062
ld: (Warning) did not generate an output file
```

To set the environment variable LPATH properly, see "HP-UX Environment Settings" in Chapter 1.

Useful Macro Definitions for PCL and FCL

Like the TIS, PCL and FCL have macro definitions included with the calling functions and error branch procedures. The macro definitions can be specified using all uppercase characters for the function name. The macro definitions are defined in the header file `pcs/wafer.h`. For example, you can use the following statement:

```
  .   .   .   .   .   .   .   .
W_START_NUMBER(1,1);
  .   .   .   .   .   .   .   .
```

The C compiler replaces the "W_START_NUMBER" string with the following strings:

```
if ((w_start_number(chip,module)) == -1)
wafer_error_handler(__FILE__, __LINE__, "W_START_NUMBER(chip,module)")
```

"`__FILE__`" and "`__LINE__`" are replaced with the file name and line number where the "W_START_NUMBER" is found by the C compiler. If an error is detected in the `w_start_number` function, the `wafer_error_handler` function is called with the source file name, line number, and macro name. The `wafer_error_handler` function reports the following error message and terminates the calling process:

```
probing.c(18): PCL/FCL ERROR 31   w_get() is not declared.
Error call: W_START_NUMBER(1,1)
```

You can easily find where an error occurs and what the error is.

You can modify the Wafer_error_handler function for your needs. The modified Wafer_error_handler function is attached in your program using the following template:

```
void wafer_error_handler(file, line, call)
char *file, *call;
int line;
{
    extern int w_errno, p_errno, errno;
  .   .   .   .   .   .   .   .
< user defined routine >
  .   .   .   .   .   .   .   .
}
```

The source file of the Wafer_error_handler function is `/usr/pcs/src/wafer/waferrh.c`. You can modify and compile the `waferrh.c` file and use the Wafer_error_handler function as your library. See Appendix A.

Probing Control Library (PCL)

The probing control library (PCL) is a set of functions for controlling an automatic wafer prober. These functions are used with the probing pattern data file created by PPG. The declarations and definitions of the PCL functions are stored in the header file `wafer.h` in the `/usr/include/pcs` directory.

PCL provides functions such as retrieving a probing pattern data file, designating an initial probing position, specifying a chip and test module to be probed, and reporting the present probing position. PCL functions are used in the same way as TIS functions, and make it easy to control the prober.

PCL Programming

Most programs written for sequential probing require only three main PCL functions: W_get, W_start_number, and W_move_next functions. These three PCL functions are used more often than any others, and provide you with the easiest method for controlling a wafer prober from a program. This paragraph describes how to create prober control programs from the various PCL functions by making extensive use of sample programs. For complete details on the syntax and descriptions of each PCL function, refer to the *Programming Reference for C Library* manual.

When writing probing control programs that contain PCL functions, be aware that there are certain restrictions with which you can use prober driver functions as follows:

- The P_move function (prober driver) cannot be used in a program that contains PCL functions.
- The P_scale and P_orig functions (prober driver) cannot be used in a program that includes a W_start_number or W_start_xy PCL function, because these PCL functions include functions identical to P_scale and P_orig functions.
- If the P_down function (prober driver) is used in a program that contains PCL functions, raise the wafer chuck before any PCL functions execute.
- If the P_imove function (prober driver) is used in conjunction with any PCL functions, the wafer chuck must be returned to its previous position before any PCL functions execute.

When you create a program using the PCL functions, the header file "`wafer.h`" must be included in the program, for example, as follows:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>
   .   .   .   .   .   .   .
   .   .   .   .   .   .   .
```

Note that the program must include the header file `prober.h` as well as `wafer.h` when using the PCL functions.

Primary Prober Control Programming

By using C programming language statements and the W_get, W_start_number, and W_move_next PCL functions, prober control programs can be written with ease. These three PCL functions perform the following tasks:

W_get — Loads probing pattern data file created by PPG. The synopsis is:

```
int w_get(filename)
char *filename;
```

W_start_number — Specifies the first probing position. That is, the present position corresponds to the first probing position when this function is executed. The synopsis is:

```
int w_start_number(chip_number, module_number)
int chip_number, module_number;
```

W_move_next — Moves the probing position to the next chip or module in incremental order. The synopsis is:

```
int w_move_next()
```

Figure 6-2 shows a sample program.

In this sample program, a hypothetical probing pattern data file named PP_DATA (see Chapter 6 in HP 4062C/UX *System Operation*) is used to perform a wafer measurement. The specified wafer contains 100 chips, and one module on each chip is specified for probing.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    INST fd;
    int i;

    if ((fd = open_tis_hpib_fd()) == -1) error_rep();          making the HP-UX communicate with
    set_tis_hpib_fd(fd);                                       instruments
    if (prober_init("EG1034X", 0, 0) == -1) p_error_rep();     initializing the wafer prober

    if (init_system() == -1) error_rep();                      initializing the system instruments
    if (lock_system(0) == -1) error_rep();

    printf("Align the probing position at the first chip, ");  Operator must align the probing position.
    printf("and set the prober to AUTO mode.\n");
    printf("Then press [Return].\n");
    while (getchar() != '\n');

    if (w_get("PP_DATA") == -1) w_error_rep();                 loading a probing pattern data file
    if (w_start_number(1, 1) == -1) w_error_rep();             setting the first probing chip
    for (i = 1; 1 <= 100; i++) {
        measure();                                             measuring
        if (w_move_next() == -1) w_error_rep();                moving to the next chip
    }
}
measure();
{
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
}
error_rep();
{
    fprintf(stderr, "tis_errno = %d,    ", tis_errno);
    exit(-1);
}
p_error_rep();
{
    fprintf(stderr, "p_errno = %d,    ", p_errno);
    exit(-1);
}
w_error_rep();
{
    fprintf(stderr, "w_errno = %d,    ", w_errno);
    fprintf(stderr, "p_errno = %d,    ", p_errno);
    exit(-1);
}
```

Figure 6-2. PCL Programming Example: Primary Prober Control

Using MACRO definitions of PCL functions:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    INST fd;
    int i;

    OPEN_TIS_HPIB;
    PROBER_INIT("EG1034X", 0, 0);

    INIT_SYSTEM();
    LOCK_SYSTEM(0);

    printf("Align the probing position at the first chip, ");
    printf("and set the prober to AUTO mode.\n");
    printf("Then press [Return].\n");
    while (getchar() != '\n');

    W_GET("PP_DATA");
    W_START_NUMBER(1, 1);

    for (i = 1; 1 <= 100; i++) {
        measure();
        W_MOVE_NEXT();
    }
}
measure();
{
  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .
}
```

PCL Programming Example: Primary Prober Control Using Macro Definitions

Confirming the Present Probing Position

You can use the W_pos_number function to verify the chip and test module presently being probed. The synopsis of the W_pos_number function is as follows:

```
int w_pos_number(chip_number, module_number)
int *chip_number, *module_number;
```

This function returns the present chip number to the first parameter *chip_number*, and the present module number to the second parameter *module_number*.

For example, this function is used to show the present probing position. Figure 6-3 shows an example program to satisfy the following restrictions:

- The wafer has 100 chips and the chip has 3 modules.
- The probing starts at chip 1 and module 1.
- The probing position moves sequentially.
- The probing stops at chip 5 and module 4.
- To test module n, `test`n is used.

■ The present probing position is displayed.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    INST fd;
    int i;

    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);
    if (prober_init("EG1034X", 0, 0) == -1) error_rep();

    if (init_system() == -1) error_rep();
    if (lock_system(0) == -1) error_rep();

    printf("Align the probing position at the first chip, ");
    printf("and set the prober to AUTO mode.\n");
    printf("Then press [Return].\n");
    while (getchar() != '\n');

    if (w_get("PP_DATA") == -1) error_rep();
    if (w_start_number(1, 1) == -1) error_rep();

    do {
        if (w_pos_number(chip, module) == -1) error_rep();
        printf("Chip = %d,   Module = %d\n", chip, module);
        switch (module) {
            case 1: test1();
                    break;
            case 2: test2();
                    break;
            case 3: test3();
                    break;
            default:
        }
        if (w_move_next() == -1) error_rep();
    } while (chip != 5 || module != 3);
    printf("Test End !\n");
}
void test1()
{
  .  .  .  .  .  .  .  .
}
void test2()
{
  .  .  .  .  .  .  .  .
}
void test3()
{
  .  .  .  .  .  .  .  .
}
void error_rep()
{
  .  .  .  .  .  .  .  .
}
```

Figure 6-3. PCL Programming Example: Showing the Present Probing Position

"Free" Probing

You can use the following function to specify the probing position:

W_move_number Moves the prober chuck to the test module designated by the parameters. The synopsis is:

```
int w_move_number(chip_number, module_number)
int chip_number, module_number;
```

The first parameter *chip_number* specifies the chip number, and the second parameter *module_number* specifies the module number.

Figure 6-4 shows a sample program.

In this sample program, assume that the specified wafer has 100 chips and the chip has four modules. This sample program specifies module 1 on each chip for probing.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    INST fd;
    int chip;

    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);
    if (prober_init("EG1034X", 0, 0) == -1) error_rep();

    if (init_system() == -1) error_rep();
    if (lock_system(0) == -1) error_rep();

    printf("Align the probing position at the first chip, ");
    printf("and set the prober to AUTO mode.\n");
    printf("Then press [Return].\n");
    while (getchar() != '\n');

    if (w_get("SAMPLE_W1") == -1) error_rep();
    if (w_start_number(1, 1) == -1) error_rep();

    for (chip = 1; chip <= 100; chip++) {
        if (w_move_number(chip, 1) == -1) error_rep();          specifying the probing position
        measure();
    }
}
measure()
{
  . . . . . .
}
error_rep()
{
  . . . . . .
}
```

Figure 6-4. PCL Programming Example: "Free" Probing Control

Probing Control Using X- and Y-chip Coordinates

You can specify chips for probing by using X- and Y-coordinates instead of chip numbers. *Chip_x* and *chip_y* are the X- and Y-coordinates of the chips set in PPG. The following three functions use the X- and Y-chip coordinates to control the prober:

W_start_xy — Specifies the first probing position using the X- and Y-coordinates, and functions the same as the W_start_number function. The synopsis is:

```
int w_start_xy(chip_x, chip_y, module_number)
int chip_x, chip_y, module_number;
```

W_pos_xy — Returns the present position (X- and Y-coordinates) and present module number, and functions the same as the W_pos_number function. The synopsis is:

```
int w_pos_xy(chip_x, chip_y, module_number)
int *chip_x, *chip_y, *module_number;
```

W_move_xy — Specifies the probing position (X- and Y-coordinates) in an order not related to the probing sequence set in the probing pattern data. The synopsis is:

```
int w_move_xy(chip_x, chip_y, module_number)
int chip_x, chip_y, module_number;
```

Figure 6-5 shows a sample program.

In this sample program, assume that the specified wafer has 30 chips and the chip has four modules. This sample program specifies module 1 and 2 on the chips at coordinates (0,3), (0,0), and (0,−3) for probing.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    INST fd;
    int chip_y, x, y, module;

    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);
    if (prober_init("EG1034X", 0, 0) == -1) error_rep();

    if (init_system() == -1) error_rep();
    if (lock_system(0) == -1) error_rep();

    printf("Align the probing position at the top center chip, ");
    printf("and set the prober to AUTO mode.\n");
    printf("Then press [Return].\n");
    while (getchar() != '\n');

    if (w_get("SAMPLE_W1") == -1) error_rep();
    if (w_start_xy(0, 3, 1) == -1) error_rep();

    for (chip_y = 3; chip_y >= -3; chip_y -= 3) {
        if (w_move_xy(0, chip_y, 1) == -1) error_rep();
        do {
            if (w_pos_xy(&x, &y, &module) == -1) error_rep();
            printf("CHIP(%d,%d)  MODULE=%d\n", x, y, module);
            switch (module) {
                case 1: test1();
                        break;
                case 2: test2();
                        break;
                default:
            }
            if (module != 4)
                if (w_move_next() == -1) error_rep();
        } while (module != 4);
    }
}
void test1()
{
  .  .  .  .  .  .  .  .
}
void test2()
{
  .  .  .  .  .  .  .  .
}
void error_rep()
{
  .  .  .  .  .  .  .  .
}
```

Figure 6-5. PCL Programming Example: Probing Control Using X- and Y-Chip Coordinates

Other PCL functions

In addition to the previously explained functions, the following functions are also supplied:

W_return_limit — Returns the maximum and minimum value of the X-Y chip coordinates from the loaded probing pattern data. The synopsis is:

```
int w_return_limit(min_x_chip, max_x_chip,
                   min_y_chip, max_y_chip)
int *min_x_chip, *max_x_chip;
int *min_y_chip, *max_y_chip;
```

W_return_xy — Returns the X-Y chip coordinates converted from the chip number according to the loaded probing pattern data. The synopsis is:

```
int w_return_xy(x, y, number)
int *x, *y, number;
```

The converted X-Y coordinates are stored in the x and y, respectively.

W_return_number — Returns the chip number converted from the X-Y chip coordinates according to the loaded probing pattern data. The synopsis is:

```
int w_return_number(x, y)
int x, y;
```

W_total_chip — Returns the total number of the chips for probing according to the loaded probing pattern data. The synopsis is:

```
int w_total_chip()
```

For more details on these functions, see the *Programming Reference for C Library*.

File Creation Library (FCL)

The file creation library (FCL) is a set of functions used to create data files for wafer maps and statistical graphics. FCL created data files are compatible with BSDM. The declarations and definitions of the FCL functions are stored in the header file `wafer.h` in the `/usr/include/pcs` directory. For an explanation of BSDM, see Chapter 7 "Statistical Graphics" and Appendix B "BSDM and Statistical Graphics" in the HP 4062C/UX *System Operation*.

The FCL can also create a DOS text file that includes the measured data. The W_file_type function specifies the data file type: BASIC BDAT (BSDM compatible) or DOS text file.

The FCL functions store the following items of the measured data in a file:

File name	Is a file name for the stored data.
Title	Is a title of the data file. This is also used as a comment, for example, "BANZAI-chip, Lot No. 1738, 5/15/89".
Number of observation (No)	Is the number of measurements for each device parameter such as h_{FE}. For example, the No concurs with the number of chips for probing.
Number of Variables (Nv)	Is the number of measurement items corresponding to the variables.
Data set	Is a group of data similar to the two-dimensional array as follows:

File name and title (comment)

	Observation			
	1	2	· · · · ·	*No*
Variable 1	data 1	data 2	· · · · ·	data *No*
Variable 2	data 1	data 2	· · · · ·	data *No*
⋮		⋮		⋮
⋮		⋮		⋮
Variable Nv	data 1	data 2	· · · · ·	data *No*

You must be aware of the following rules for the usage of the data files:

- Data files for Wafer Maps:

  The data number must concur with the probed chip number. When the W_put_data function is used, these numbers automatically concur. For example, the following table can be supposed:

	Observation			
	chip 1	chip 2	· · · · ·	chip 100
Vth	0.653	0.677	· · · · ·	0.701
h_{FE}	105	156	· · · · ·	182

- Data files for Time Plots (Control Charts):

  The data number must concur with the plot order. For example the following table can be supposed:

	Observation			
	wafer 1	wafer 2	· · · · ·	wafer 24
Vth (average)	0.685	0.673	· · · · ·	0.691
h_{FE} (average)	163	158	· · · · ·	166

- Data files for Histograms and Scattergrams:

  No special attention is needed.

FCL Programming

As Figure 6-6 shows, there are three ways to create a measurement data file using FCL functions. You can get the data from the created data file by using the FCL functions. Each method is described below:

- W_put_data and W_save_data (see Figure 6-7 for an example):

  This method can be used only when PCL subprograms are used to control the wafer prober. This method is ideal for creating measurement data files for wafer mapping because measurement data are automatically arranged in chip number order, not measurement order. Data files created this way can also be used for creating histograms and scattergrams.

- W_put_array and W_save_data (See Figure 6-8 for an example):

  This method is used for generating measurement data files for statistical graphics. With this method, one-dimensional user-specified arrays are filled with measurement data, transferred to the variable area allocated by the FCL function, and assigned variable names. The data in the variable area are then combined to create a measurement data file. If W_put_array function is used in a measurement program, W_put_data function cannot be used, and vice versa.

- W_build_file (Figure 6-9 for an example):

  This function creates and stores multiple parameter measurement data files for statistical graphics. The variable names used as parameters in this function correspond to each row of the data array. The W_build_file function does not allocate the variable area to keep the measured data.

- W_file_type

  This function specifies the data file type: BASIC BDAT or DOS text file. The W_save_data and W_build_file functions save a data file according to this file type setting. The default setting is the BASIC BDAT file.

- W_read_file

  This function reads the data file created by the FCL functions, and stores the data into a specified structured variable. The header file `wafer.h` contains the structure definition used

to store the data created by the FCL functions. This function can read any other BSDM data file.

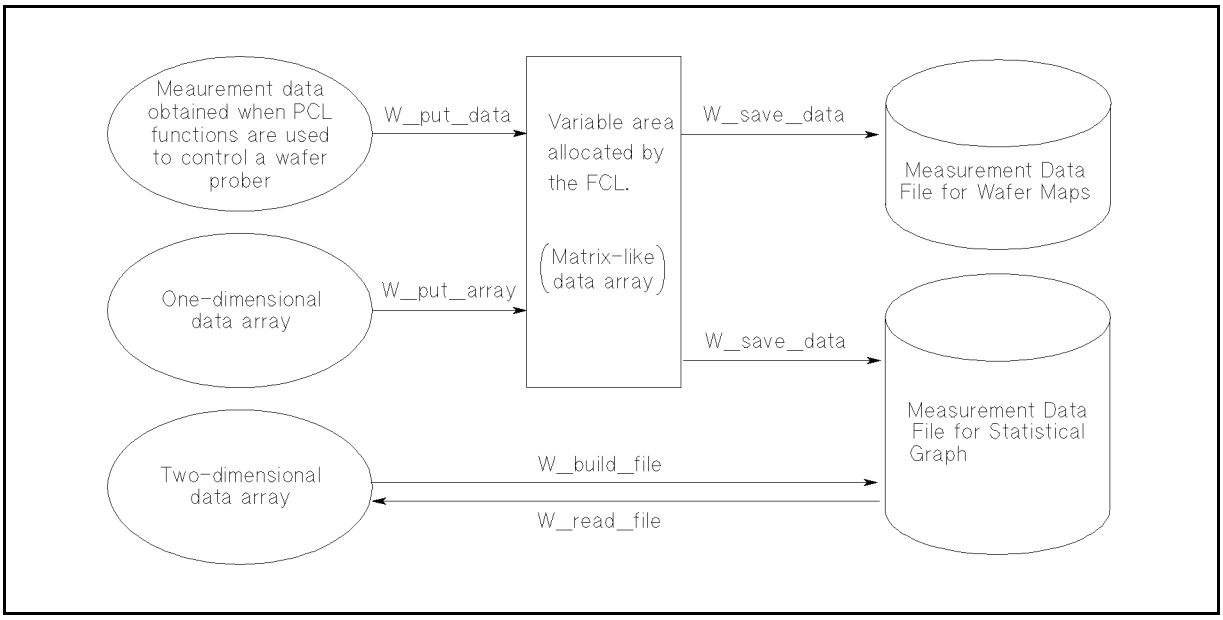


Figure 6-6. FCL Functions

When you create a program using the FCL functions, the header file "`wafer.h`" must be included in the program, for example, as follows:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>
   .   .   .   .   .   .   .
   .   .   .   .   .   .   .
```

Creating Data Files for Wafer Maps

The W_put_data and W_save_data functions are primarily used to create data files for wafer mapping. The synopsis of these functions are as follows:

```
int w_put_data(variable_name, data)
char *variable_name;
double data;

int w_save_data(filename, title)
char *filename, *title;
```

The following explains these functions using examples:

- `w_put_data("hFE", hfe_data);`

  The W_put_data function transfers the measured data (in the variable `hfe_data`) into a specified memory area with the variable name "`hFE`".

- `w_save_data("DATA_W1","DATA OF WAFER 1");`

  The W_save_data function transfers all of the data saved by W_put_data function into a data file named "DATA_W1", with the comment "DATA OF WAFER 1".

In this sample program, you measure h_{FE} at module 1 and sheet resistance at module 2.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    INST fd;
    int total_chips, i;
    int chip, module;

    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);
    if (prober_init("EG1034X", 0, 0) == -1) error_rep();

    if (init_system() == -1) error_rep();
    if (lock_system(0) == -1) error_rep();

    printf("Align the probing position at the top center chip, ");
    printf("and set the prober to AUTO mode.\n");
    printf("Then press [Return].\n");
    while (getchar() != '\n');

    if (w_get("SAMPLE_W1") == -1) error_rep();
    total_chips = w_total_chip();
    if (w_start_number(1,1) == -1) error_rep();

    for (i = 1; i <= total_chips; i++) {
        if (w_pos_number(&chip, &module) == -1) error_rep();
        switch (module) {
            case 1: meas_hfe();
                    break;
            case 2: meas_r();
                    break;
            default:
        }
        if (w_move_next() == -1) error_rep();
    }
    if (w_save_data("DATA_W1", "DATA OF WAFER 1") == -1) error_rep();
}
void meas_hfe()
{
  .  .  .  .  .  .  .  .  .
}
void meas_r()
{
  .  .  .  .  .  .  .  .  .
}
void error_rep()
{
  .  .  .  .  .  .  .  .  .
}
```

Figure 6-7. FCL Programming Example: Data Logging for Wafer Mapping

Creating Data Files for Statistical Graphics

The W_put_array and W_build_file functions are mainly used to create data files for statistical graphics. An example of how each one of these subprograms might be used is as follows:

- `w_put_array("hFE", data1, 20);`

  In this example, the W_put_array function transfers the measurement data contained in a one-dimensional array `data1`, to the specified variable area under the variable name "h_{FE}". To save the data, the W_save_data function is used the same way as the W_put_data function.

- `w_build_file("DATA_W2", data, 100, "Data for Statistics", "hFE", "Vth", NULL);`

  In this example, the W_build_file function stores the measurement data of two-dimensional array `data` into data file "`DATA_W2`" (with variable names "h_{FE}" and "Vth" assigned to row 1 and row 2 of the data file, respectively), with the comment "Data for Statistics".

This sample program measures h_{FE} and Vth for 20 devices, and saves the measured data using the W_put_array and W_save_data functions.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    INST fd;
    int i;
    double hfe[20], vth[20];

    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);

    if (init_system() == -1) error_rep();
    if (lock_system(0) == -1) error_rep();

    for (i = 1; i <= 20; i++) {
        test(&hfe[i], &vth[i]);
    }
    if (w_put_array("hFE", hfe, 20) == -1) error_rep();
    if (w_put_array("Vth", vth, 20) == -1) error_rep();
    if (w_save_data("DATA1", "DATA FOR STATISTICS") == -1) error_rep();
}
void test(hfe, vth);
double *hfe, *vth;
{
   .  .  .  .  .  .  .  .  .
}
void error_rep()
{
   .  .  .  .  .  .  .  .  .
}
```

Figure 6-8.
FCP Programming Example: Creating a Data File for Statistical Graphics Using W_put_array and W_save_data

This sample program measures h_{FE} and Vth for 20 devices, and saves the measured data using the W_build_file function.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    char *title = "DATA FOR STATISTICS";
    INST fd;
    int i;
    double hfe, vth, data[2][20];

    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);

    if (init_system() == -1) error_rep();
    if (lock_system(0) == -1) error_rep();

    for (i = 1; i <= 20; i++) {
        test(&hfe, &vth);
        data[1][i] = hfe;
        data[2][i] = vth;
    }
    if (w_build_file("DATA1", data, 20, "hFE", "Vth", NULL) == -1) error_rep();
}
void test(hfe, vth)
double *hfe, *vth;
{
   .  .  .  .  .  .  .  .  .
}
void error_rep()
{
   .  .  .  .  .  .  .  .  .
}
```

Figure 6-9.
FCL Programming Example: Creating a Data File for Statistical Graphics Using W_build_file

Reading a BSDM Format Data File

The W_read_file function is used to read the BSDM data file, which is created by the FCL functions, such as W_build_file. The synopsis of this function is as follow:

```
int w_read_data(filename, data_struct)
char *filename;
struct fcl_struct *data_struct;
```

The structure tag `fcl_struct` is declared in the header file `wafer.h`. You must declare the structure variable of the `fcl_struct` in the program. Figure 6-10 shows a sample program reading a BSDM data file using the W_read_data function.

This sample program reads data from the BSDM data file "DATA1", and shows the title and the number of data.

```
#include <stdio.h>
#include <pcs/wafer.h>

main()
{
    int i, j;
    struct fcl_struct &data;

    if (w_read_file("DATA1", data) == -1) error_rep();
    printf("Title: %s\n", data->title);
    printf("Number of variables: %f\n", data->var_num);
    printf("Number of observation: %f\n", data->obs_num);
    printf("Variable name and data:\n");
    for ( i = 0 ; i < data->var_num ; i++) {
        printf("[ %s ]\n", var_name[i]);
        for ( j = 0 ; j < data->obs_num ; j++ )
            printf("%f ", data->data[i][j]);
    }
void error_rep()
{
  .  .  .  .  .  .  .  .
}
```

Figure 6-10. FCL Programming Example: Reading a BSDM Data File Using W_read_file

Creating DOS Data Files

The W_save_data and W_build_file function can save measured data in a DOS file instead of the BDAT file formatted for the BSDM. The DOS file can be retrieved by any database and data analysis software because it is one of the ASCII text files.

The W_file_type function specifies the type of the data file created by the W_save_data and W_build_file functions. The data file type is set to the BDAT file on initialization.

The following example sets the data file type to DOS file and creates a DOS data file using the W_save_data function.

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/wafer.h>
#include <sicl.h>

main()
{
    INST fd;
    int i;
    double hfe[20], vth[20];

    if ((fd = open_tis_hpib_fd()) == -1) error_rep();
    set_tis_hpib_fd(fd);

    if (init_system() == -1) error_rep();
    if (lock_system(0) == -1) error_rep();

    for (i = 1; i <= 20; i++) {
        test(&hfe[i], &vth[i]);
    }
    if (w_put_array("hFE", hfe, 20) == -1) error_rep();
    if (w_put_array("Vth", vth, 20) == -1) error_rep();
    if (w_file_type("DOS") == -1) error_rep();
                                                    Specifying the data file type to
                                                    DOS file
    if (w_save_data("DATA1", "DATA FOR STATISTICS") == -1) error_rep();
}
void test(hfe, vth);
double *hfe, *vth;
{
    . . . . . .
}
void error_rep()
{
    . . . . . .
}
```

Figure 6-11. Sample Program: Creating a DOS Data File

A

Customizing Your Prober Driver

This appendix provides information on how to customize your prober drivers. Use the wafer prober manuals to customize or create the prober driver software.

You must customize or create the prober driver software in the following cases:

- Your wafer prober is an Electroglas 2001X/2010X/3001X/4060X/4080X that is not equipped with an AUTO LOAD, a PROFILE, or an AUTO ALIGN option.
- You use prober oriented functions, such as to return error codes generated by the prober.
- You use an unsupported wafer prober.

The source files of the prober driver software are stored in the `/usr/pcs/src/prober` directory. You can refer to the source files to customize or create the prober drivers.

The source files of the prober drivers are categorized into three types:

`prober_`*xxx*`.c`	Includes prober driver functions for each prober type. The *xxx* indicates the prober type. Note that `prober_man.c` and `prober_utl.c` include generic functions such as the Prober_init and Prober_output functions.
`proberrh.`*x*	Used for error handling routines for the prober drivers. You can modify this source file to adjust your error handler routines.
`stub_`*xxx*`.c`	Includes a program aborting routine when the necessary prober library is not linked. The library routines generated from the `stub_`*xxx*`.c` files are included in the `libhp4062.a` file.

Note

When customizing or creating the prober drivers, pay attention to the following items:

- Do not modify the source files in the `/usr/pcs/src/prober` directory. You might lose the original source files if you modify the source files in the `/usr/pcs/src/prober` directory. Create a working directory, copy the necessary source files into the working directory, and create the new prober drivers.
- You must link the proper prober drivers you create. If you link the prober drivers stored in the `/usr/pcs/lib` directory, the operation of the prober driver is not guaranteed.

Tip

You can use the `make` command to manage program development. It is especially useful to manage the revision of a source file. See *HP-UX Concepts and Tutorials: Programming Environment* for more information.

Modifing the Electroglas 2001X/2010X/3001X/4060X/4080X Wafer Prober

If your Electroglas 2001X/2010X/3001X wafer prober (the EGY wafer prober) is not equipped with an AUTO LOAD, a PROFILE, or an AUTO ALIGN option, you must modify the source file and create the library again. The wafer prober reports an error showing that the specified function is not supported if you do not modify the file.

The P_home function in the `libegy.a` activates all these functions. The P_home_egy function in the `prober_egy.c` must be modified according to the options for your EGY prober as follows:

- If the AUTO LOAD option is not furnished, delete the following lines:

```
ret = prober_output(p_eid, p_address, "LO", TERM_CRLF);
if ( ret == -1 )                /* unload wafer and load new wafer */
        return ( -1 );

ret = wait_ready_egy();         /* wait response from prober */
if ( ret == -1 )
        return ( -1 );
```

- If the PROFILE option is not furnished, delete the following lines:

```
ret = prober_output(p_eid, p_address, "PZ", TERM_CRLF);
if ( ret == -1 )                /* find wafer thickness */
        return ( -1 );

ret = wait_ready_egy();         /* wait response from prober */
if ( ret == -1 )
        return ( -1 );
```

- If the AUTO ALIGN option is not furnished, delete the following lines:

```
ret = prober_output(p_eid, p_address, "AA", TERM_CRLF);
if ( ret == -1 )                /* do alignment */
        return ( -1 );
```

Use the following procedure to modify the source file and to recreate the prober driver:

1. Create a working directory to create the prober driver again. For example, to create the `/users/customize/prober` directory, enter:

   ```
   $ mkdir /users/customize/prober
   ```

2. Copy the source file `prober_egy.c` to the working directory as follows:

   ```
   $ cp /usr/pcs/src/prober/prober_egy.c /users/customize/prober
   ```

3. Change the current directory to the working directory as follows:

   ```
   $ cd /users/customize/prober
   ```

4. Modify the source file `prober_egy.c` using an editor, such as the `vi` editor.

5. Compile the modified source file with the `-c` option as follows:

   ```
   $ cc -c prober_egy.c
   ```

   The object file `prober_egy.o` is created.

When you compile the test program, for example, use the following command line:

```
$ cc test_prog.c -o test_prog -lprober /users/customize/prober/prober_egy.o
-lhp4062 -lm -lsicl -lcl
```

The `prober_egy.o` must be specified before the `-lhp4062` options.

Adding Prober Oriented Functions

You can create new functions to control prober oriented functions, such as a function that reads the error code of the wafer prober. These functions can be created in the source file of the test program, or can be created in a source file separate from the source file of the test program. This section provides how to create the functions as a library.

To develop the new prober driver functions, create a working directory and create a source file in the working directory. When the new functions are created as a library, a header file containing the external declarations of the new functions must also be created.

For example, when the new function "P_get_error" is created in the source file "`newfunc.c`," the following procedure can be used:

1. Create a working directory to create the prober driver again. For example, to create the `/users/customize/prober` directory, enter:

   ```
   $ mkdir /users/customize/prober
   ```

2. Change the current directory to the working directory as follows:

   ```
   $ cd /users/customize/prober
   ```

3. Create the source file `newfunc.c` using an editor, such as the `vi` editor. The contents of the source file `newfunc.c` are, for example, as follows:

   ```
   /*****************************************************************/
   /* P_get_error gets an error code from the TSK prober.  This     */
   /* function must be executed after the serial poll is executed.  */
   /*                                                               */
   /* Parameter:                                                    */
   /*    eid: The session ID of the HP-IB interface                 */
   /*     ba: HP-IB primary address of the wafer prober             */
   /*   buff: The pointer to a buffer storing the returned error    */
   /*         code.   The buffer size must be more than 7.          */
   /*   size: The size of the buffer                                */
   /*****************************************************************/
   #include <pcs/prober.h>
   #include <sicl.h>

   int p_get_error(eid, ba, buff, size)
   INST eid;
   int ba, size;
   char *buff;
   {
       int ret;

       ret = prober_output(eid, ba, "E", 0);
       if (ret == -1) return (-1);

       ret = prober_enter(eid, ba, buff, size);
       return (ret);
   }
   ```

4. Compile the source file `newfunc.c` with the compile only option (`-c`) as follows:

```
$ cc -c newfunc.c
```

If no error is detected, the object file `newfunc.o` is created.

You must also create the header file "newfunc.h" that includes an external declaration of the P_get_error function as follows:

```
/* Header file: newfunc.h */

#ifdef __STDC__

extern int p_get_error(int, int, char *, int);

#else /* __STDC__ */

extern int p_get_error();

#endif /* __STDC__ */
```

Note that the header file above supports both the ANSI and K&R standard of C programming language.

When you use the new P_get_error function, the source file of the test program must include the header file `newfunc.h` as well as other header files. The following shows an example:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/prober>
#include "/users/customize/prober/newfunc.h"
#include <sicl.h>

main()
{
  .   .   .   .   .   .   .   .
  .   .   .   .   .   .   .   .
}
p_error_rep()
{
    INST eid;
    int ba;
    char err_msg[10];

    eid = prober_get_eid();
    ba = prober_get_ba();
    if (p_get_error(eid, ba, err_msg, 9) == -1) {
        fprintf(stderr, "Prober driver error: %d\n", p_errno);
        exit(-1);
    }
    err_msg++;
    fprintf(stderr, "PROBER ERROR: %s\n", err_msg);
    exit(0);
}
```

When the source file above is compiled, the following command line is used:

```
$ cc test_prog.c -o test_prog /users/customize/prober/newfunc.o -lprober -l
tsk -lhp4062 -lm -lsicl -lcl
```

Creating a Prober Driver for an Unsupported Wafer Prober

For a prober driver of an unsupported wafer prober, the prober model name "CUSTOM" is reserved. You can use this model name for an unsupported wafer prober.

To create a new prober driver and to use it with the PCL functions, the specifications of the new prober driver must be consistent with the other prober drivers supplied in the HP 4062UX software. For this reason, a template file named "`prober_cst.c`" is supplied in the `/usr/pcs/src/prober` directory. You can use the template file `prober_cst.c` to create a new prober driver.

Prober Driver Specifications

In this paragraph, the prober driver specifications for an unsupported wafer prober are provided.

First, you must search the prober control commands that satisfy the following operation in the prober manual:

- Moves the X-Y stage to a position specified by absolute X-Y coordinates.
- Lowers the chuck.
- Raises the chuck.
- Moves the X-Y stage to the home position.
- Triggers the inker.

You must also look for the meanings of the status bytes for the service request (SRQ). This status byte may show the prober status or error status.

The following functions must be created:

- P_home_custom
- P_up_custom
- P_down_custom
- P_move_custom
- P_imove_custom
- P_ink_custom
- Prober_status_custom
- Prober_reset_custom

For the specifications of the above functions, see the *Programming Reference for C Library* manual.

The following explains the internal variables used in the prober drivers:

`p_remote`	Integer variable. This variable shows the prober status: Remote (= 1) or Local (= 0).
`p_onwafer`	Integer variable. This variable shows whether or not the probing position is above. If the probing position is above the wafer, the *p_onwafer* is true (= 1). Otherwise, the *p_onwafer* is false (= 0).
`p_lastwafer`	Integer variable. This variable shows whether or not the wafer on the chuck is the last one. If the wafer is the last one, the *p_lastwafer* is true (= 1). Otherwise, the *p_lastwafer* is false (= 0).

`p_xsize` and `p_ysize` — Double variables. These variables keep the X- and Y- index values set by the P_scale function.

`p_xpos` and `p_ypos` — Double variables. These variables keep the present probing position in the index values.

You can modify the source file `prober_cst.c` according to the specifications above. Proceed to the next paragraph.

Modifying and Compiling the Source File

The following procedure provides a sample to modify the template file `prober_cst.c` and to compile the source file:

1. Create a working directory to create the new prober driver. For example, to create the `/users/customize/prober` directory, enter:

   ```
   $ mkdir /users/customize/prober
   ```

2. Copy the source file `prober_cst.c` to the working directory as follows:

   ```
   $ cp /usr/pcs/src/prober/prober_cst.c /users/customize/prober
   ```

3. Change the current directory to the working directory as follows:

   ```
   $ cd /users/customize/prober
   ```

4. Modify the source file `prober_cst.c` using an editor, such as the `vi` editor. The specifications for the new prober driver are considered in the previous paragraph.

5. Compile the modified source file with the `-c` option as follows:

   ```
   $ cc -c prober_cst.c
   ```

   The object file `prober_cst.o` is created.

Using the Prober Driver for an Unsupported Wafer Prober

In the source file of the test program, you can use the prober driver for an unsupported wafer prober in the same way as other prober drivers. But you must use the prober type name "CUSTOM" in the Prober_init function as follows:

```
#include <stdio.h>
#include <pcs/tis.h>
#include <pcs/tis.h>

main()
{
    int err;
      .   .   .   .   .   .   .   .
    err = prober_init("CUSTOM", 25, 9);
      .   .   .   .   .   .   .   .
}
```

In the example above, it is assumed that the unsupported wafer prober is connected to an HP-IB interface in which the logical unit number is 25, and that the HP-IB primary (bus) address of the unsupported wafer prober is set to 9.

Tip

You can register the unsupported wafer prober in the SYSCONFIG file in the `/usr/pcs/etc` directory, for example, as follows:

```
   .   .   .   .   .   .   .   .
1150 ! WPS   APM7000A   2505
1160 ! WSP   SP900B     2508
1170 ! WSP   SP1000B    2508
1180 ! WSP   SP1100B    2508
1100 ! WSP   CUSTOM     2509    ⇐
```

To compile the source file of the test program, use the following command line:

```
$ cc test_prog.c -o test_prog -lprober /users/customize/prober/prober_cst.o
-lhp4062 -lm -lsicl -lcl
```

Note that the `prober_cst.o` must be specified before the `-lhp4062` options.

Index