Appendix A Sample Programs

A.1 Overview

The easiest way to learn about a new software resource is to look at some simple, straightforward working code. Kernighan and Ritchie's C tutorial begins with a program that types "Hello, world."

The term "hello-world program" has come to mean a sample or demo program that helps new users take the first few steps on a new system without resorting to trial and error.

Here we offer hello-world programs for C, FORTRAN, Pascal, Lisp, Modula-2, SunView, NeWS, Core, CGI, GKS,and PHIGS. Each section gives one or more sample programs, with instructions on how to compile and execute the code.

We assume that the sample programs are stored in files with specific names. The name of each program file is surrounded by asterisks, before each listing.

A.2 C

```
/***********
 * hello.c *
 ***********/
 * C program to print "Hello, world."
 *
 */
main()
{
     printf("Hello, world.\n");
}
```

This is a simple C program above that prints out "Hello, world."

There is one function, called *main*. The empty parentheses after the name indicate that *main* has no input parameters. (Input values to the main routine are the parameters the user types on the command line when invoking the program. For more information see *The C Programming Language* (in the index under *argc* and *argv*) by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.)

The body of *main* is contained between braces. There is one command: a call to *printf*, terminated by a semicolon. For more information on *printf*, see the *Sun UNIX Interface Reference Manual*, Section 3S. That section describes the standard I/O library, and if you're just getting started with C, we recommend that you also study the entries for *fopen*, *fclose*, *getc*, *gets*, *putc*, and *puts*.

To compile the program, type:

```
% cc hello.c -o hello
```

The C compiler is called cc. C source filenames must end with *.c*. Typically the object name is the source filename but without using the *-o* compiler option the object name is *a.out*. The *-o* compiler option is followed by a string that is the object name.

For more information on the C compiler, refer to the Sun *UNIX Interface Reference Manual*, Section 1, under *cc*. For additional information on compiling programs with more than one source file, refer to Section 1 under *make*.

A.3 FORTRAN

```
C     ********************
C     * fortran_sample.f *
C     ********************
      program sample
C This is a comment line.
      print *, 'Greetings from a FORTRAN program.'
      end
```

This simple FORTRAN program above that types out "Greetings from a FORTRAN program."

The program consists of four lines: the program declaration, a comment, the print command, and the "end" line.

To compile the program, type:

```
% f77 fortran_sample.f -o fortran_sample
```

The compiler is called *f77*. FORTRAN source filenames must end with the *.f* suffix. The default object name is *a.out*, but the *-o* option overrides that with the name following "-o" For more information on the FORTRAN compiler, see the *Sun UNIX Interface Reference Manual*, Section 1, under *f77*.

For more information on FORTRAN in the Sun environment, please see Sun's *FORTRAN Programmer's Guide.*

A.4 Pascal

```
/*****************
 *pascal_sample.p *
 *****************/
program sample(output);
begin
     writeln('Greetings from a Pascal program.')
end.
```

This simple Pascal program types out "Greetings from a Pascal program."

The program consists of four lines: the program declaration, the program's begin line, the command that writes the message, and the program's end line.

To compile the program, type:

```
% pc pascal_sample.p –o pascal_sample
```

The compiler is called pc. Pascal source filenames must end with the *.p* suffix. The default object name is *ahe*. The *-o* option overrides that with the name following *-o*. For more information on the Pascal compiler, see the *Sun UNIX Interface Reference Manual*, Section 1, under *pc*. For general information on Pascal in the Sun environment, please see *Pascal Programmer's Guide*.

A.5 Modula-2

```
(************************
 * modula–2_sample.mod   *
 ************************)
MODULE hello;
FROM InOut IMPORT WriteString, WriteLn;
BEGIN
     WriteString  ("Hello, World!");
     WriteLn
END hello.
```

The simple Modula-2 program types out "Hello, World!" The program consists of six lines: the program declaration, declaration of output primitive, BEGIN statement, printing of the string, a linefeed, and the END statement.

To compile the program, type:

```
% m2c –e modula–2_sample modula–2_sample.mod
```

The compiler is called *m2c*. Modula-2 source filenames end with a *.mod* suffix. For simple programs such as this example, the compile and link steps may be combined. For more information on the Modula-2 compiler, see the *Sun UNIX Interface Reference Manual*, Section 1, under *m2c*. For general information on Modula-2 in the Sun environment, please see *Using Modula-2 on the Sun Workstation*.

A.6 Assembly Languages

It is generally not a good practice to program at this level because it severely restricts portability. Also, Sun's compilers can efficiently generate optimized code at different levels, eliminating the need for assembly language programming. However, for those who still like to program in assembly, Sun supports calling between assembly and other high-level languages. Between Sun's different systems, there are three instruction sets: Motorola 680X0, Intel 80386 for the new Sun386*i*, and Sun's SPARC. In the following examples, a high-level language calls an assembly routine that increments the value of a variable by one that is passed to it.

A.6.1 Motorola 68020; Pascal Calls Assembly

main.p:

```
program main(input, output);
var
     x:integer;
procedure assfunc(var x:integer);
* declare the assembly routine *)
external c;
begin

     x := 1000;
     writeln('x before call = ', x);
     ssfunc(x);
     writeln('x after call = ', x);
end.
```

assfunc.s:

```
.globl        _assfunc
_assfunc:
     link a6, #0
     movl a6@(8), a0
     movl a0@, d0
     ovl  d0, a0@
     unlk a6
     rts
```

To compile this example, type:

```
% pc main.p assfunc.s -o executable
```

A.6.2 Intel 80386; C Calls Assembly

main.c:

```
include <stdio.h>
main()
{
     int x;

     x = 1000;
     printf("x before call = %d\n", x);
     assfunc(&x);
     printf("x after call = %d\n", x);
}
```

assfunc.s:

```
.file     "assfunc.s"
.globl     assfunc
assfunc:
     pushl%ebp
     movl %esp, %ebp
     movl 8(%esp), %eax
     incl (%eax)
     leave
     ret
```

A.6.3 SPARC; FORTRAN Calls Assembly

To compile this program, type:

```
% cc main.c assfunc.s -o executable
```

main.f:

```
program main
integer x
x = 1000
print *, 'x before call = ', x
call assfunc(x)
print *, 'x after call = ', x
stop
end
```

assfunc.s:

```
     .global_assfunc_
     _assfunc_:
     ld   [%o0], %o1
     inc  %o1
     st   %o1, [%o0]
     retl
     add  %g0,0,%o0
```

To compile this program, type:

```
% f77 main.f assfunc.s
```

A.7 Sockets

```
          /*************************
          * socketsample_server.c *
          *************************/
          }
/*

         Server.c

         opens a socket in the internet domain (AF_INET) and
binds to a port (IPPORT_RESERVED + 100). It then listens on that
port. Once a connection is requested and made, the server and
client communicate with regular "read" and "write" system calls

*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void sighandler();
main()
{

        int sd, ns;
        char inbuf[256], outbuf[256];
        struct sockaddr_in sockaddr, from;
        int fromlen, rt;
        sockaddr.sin_port = IPPORT_RESERVED + 100;
        sd = socket(AF_INET, SOCK_STREAM, 0);
        if (bind(sd, (char *)&sockaddr, sizeof(sockaddr)) < 0)
        {
```

```
                printf("bind failed.\n");
                perror("bind");
                exit(1);
        }
        listen(sd, 1);
        fromlen = sizeof(from);
        ns = accept(sd, &from, &fromlen);
        close(sd);
        printf("Waiting for message ......\n");
        while(1)
        {
                if ((rt = read(ns, inbuf, sizeof(inbuf))) == 0)
continue;
                inbuf[rt] = '\0';
                printf("Server read: %s\n", inbuf);
                printf("type in your message: ");
                gets(outbuf);
                write(ns, outbuf, strlen(outbuf));
                printf("Wait for message ......\n");
        }
}
               /*************************
           * socketsample_client.c *
           *************************/
          /*

/*

          Client.c

          opens a socket in the internet domain (AF_INET) and
connects to port (IPPORT_RESERVED + 100) on a remote host, in
this example "soorya". Once this connection is established, the
client and thre remote host communicate via regular "read" and
"write" system calls

*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

main()
{
```

```
        int sd, rt;
        char outbuf[256], inbuf[256];
        struct sockaddr_in sockaddr;
        struct hostent *hp;
        if ((hp = gethostbyname("soorya")) == NULL)
        {
                printf("unknown host\n");
                exit(1);
        }
        bzero((char *)&sockaddr, sizeof(sockaddr));
        bcopy(hp->h_addr, (char *)&sockaddr.sin_addr, hp-
>h_length);
        sockaddr.sin_family = AF_INET;
        sockaddr.sin_port = IPPORT_RESERVED + 100;
        sd = socket(AF_INET, SOCK_STREAM, 0);
        if (connect(sd, &sockaddr, sizeof(sockaddr)) == -1)
        {
                printf("Connect failed.\n");
                perror("connect");
                exit(1);
        }
        for (;;)
        {
                printf("type in your message: ");
                gets(outbuf);
                write(sd, outbuf, strlen(outbuf));
                printf("Waiting for message ......\n");
                while (1)
                {
                        if ((rt = read(sd, inbuf, sizeof(inbuf)))
== 0)
                        continue;
                        inbuf[rt] = '\0';
                        printf("Client read: %s\n", inbuf);
                        break;
                }
        }
}
```

These two programs illustrate above how two processes can communicate via a socket. The first program, socketsample_server.c, listens for input on the socket and echoes to the screen whatever characters come in. The second program, socketsample_client.c, prompts the user to type in a message; this message is sent to the server via the socket.

To compile the programs, type the following two lines:

```
% cc socketsample_server.c -o socketsample_server
% cc socketsample_client.c -o socketsample_client
```

To execute the programs, first run the server in background:

```
% socketsample_server &
```

Then run the client:

```
% socketsample_client
```

The client will prompt you for messages to be sent to the server. When you are finished, type `Control-C` at the Type in your message: prompt.

A.8 Remote Procedure Calls (RPC)

```
/*********************
 * rpcsample_server.c *
 *********************

#include <rpc/rpc.h>

char *add_one();
main()
{
     /* Register the "add_one" routine */
     registerrpc(0x20000100,  1,  1,  add_one,  xdr_int,
xdr_int);

     /* Run forever */
     svc_run();
}

char *add_one(input_value)
char *input_value;
{

     int return_me;

     /* Think of input_value as a pointer to an     */
     /* integer; take what it points to, and add 1. */
     return_me = *(int *)input_value + 1;

     /* Return a pointer to return_me, cast as a    */
     /* pointer to a char instead of an int because */
```

```
    /* that's what XDR expects.                    */
    return((char *)&return_me);
}

    /*********************
     * rpcsample_client.c *
     *********************/

#include <rpc/rpc.h>

main(argc, argv)
int argc;
char *argv[];
{
    int argument, result;

    /* Check that there are command-line parameters.
*/
    if (argc != 3) {
        printf("Usage:      %s      server_hostname
input_value\n",
                argv[0]);
        exit(1);
    }
    /* Convert command-line parameter string to an
integer.  */
    argument = atoi(argv[2]);
    /* Make the rpc call.  */
    callrpc(argv[1], 0x20000100, 1, 1,
            xdr_int, &argument,xdr_int, &result);
    /* Print result.  */
    printf("Result = %d\n", result);
}
```

These two programs above show how to implement a simple RPC client/server pair.

The server registers a routine called add_one, which takes an integer input, adds one, and returns the integer result. This routine is registered with program number = 0x20000100, version number = 1, and procedure number = 1.

To compile the server, type:

```
% cc rpcsample_server.c -o rpcsample_server
```

To compile the client, type:

```
% cc rpcsample_client.c –o rpcsample_client
```

To execute the programs, first run the server on the host of your choice (it could be the one you are sitting at or any other one on the network.) To run the client, type `rpcsample_client`, followed by the name of the machine which is running the server program, followed by an integer. For example, if your server program is running on a machine named terrapin and you want to know what the next number after 11 is, type:

```
% rpcsample_client terrapin 11
```

The client program will send 11 to terrapin as a parameter of the add_one function. Terrapin will add 1 to 11 and return the result to the local machine, which will output:

```
Result = 12
```

For more information on RPC, please refer to the *Sun Network Programming Manual*. For more sophisticated RPC programming, Sun provides a code-generation tool named rpcgen. This tool is also described in the *Network Programming Manual*.

A.9 External Data Representation (XDR)

```
     /********************\
      * xdrsample_server.c *
     \*********************/

#include <rpc/rpc.h>

struct int_and_char_struct {
     int i;
     char c;
};

xdr_int_and_char(xdrsp, int_and_char_struct_ptr)
XDR *xdrsp;
struct int_and_char_struct *int_and_char_struct_ptr;
{
     if (!xdr_int(xdrsp, &int_and_char_struct_ptr->i))
          return(FALSE);
     if (!xdr_char(xdrsp, &int_and_char_struct_ptr->c))
          return(FALSE);
     return(TRUE);
}

char *add_one();

main()
{
     /* Register the "add_one" routine */
     registerrpc(0x20000101, 1, 1,
          add_one, xdr_int_and_char, xdr_int_and_char);

     /* Run forever */
     svc_run();
}

char *add_one(input_value)
char *input_value;
{
     struct int_and_char_struct return_me;
     struct int_and_char_struct *input_ptr;

     input_ptr = (struct int_and_char_struct *)
          input_value;
     return_me.i = input_ptr->i + 1;
     return_me.c = input_ptr->c + 1;
```

```
}

     /********************
     * xdrsample_client.c *
     \********************/
#include <rpc/rpc.h>

struct int_and_char_struct {
     int i;
     char c;
};

static struct int_and_char_struct arguments, results;

xdr_int_and_char(xdrsp, int_and_char_struct_ptr)
XDR *xdrsp;
struct int_and_char_struct *int_and_char_struct_ptr;
{
     if (!xdr_int(xdrsp, &int_and_char_struct_ptr->i))
          return(FALSE);
     if (!xdr_char(xdrsp, &int_and_char_struct_ptr->c))
          return(FALSE);
     return(TRUE);
}

main(argc, argv)
int argc;
char *argv[];
{
     /* Check that there are command-line parameters.
*/
     if (argc != 4) {
          printf("Usage:    %s    server_hostname    int
char\n", argv[0]);
          exit(1);
     }
     /* Store command-line parameters in struct.  */
     arguments.i = atoi(argv[2]);
     arguments.c = argv[3][0];
     /* Make the rpc call.  */
     callrpc(argv[1], 0x20000101, 1, 1,
     xdr_int_and_char, &arguments,
     xdr_int_and_char, &results);
     /* Print results.  */
     printf("Result = %d   %c\n", results.i, results.c);
}
```

These programs above illustrate the user-defined XDR facility of Remote Procedure Calls (RPCs).

Simple RPC calls, as illustrated in the last section, pass single parameters and return values. However, it is sometimes desirable to pass more complicated values, such as strings or structs. To do this, a programmer must concoct a custom XDR call out of the standard building blocks.

In our example, we pass a struct called an int_and_char_struct, which consists of one integer and one character. The client initializes this struct from the user's command line and passes it to the server. The server increments both fields of its input struct and returns the results in another int_and_char_struct.

Both the server and the client contain the routine xdr_int_and_char(), which handles translation between int_and_char_structs and XDR. The xdr_int_and_char() routine just calls xdr_int() and xdr_char() in turn, since the struct consists of an int followed by a char.

Correct programming style calls for the definition of the int_and_char_struct structure to be in a *.h header file, and for xdr_int_and_char() to be in its own module, which is linked with the server and client programs. This ensures that there is only one version of the struct definition and the xdr routine. We have neglected good style in this case to make the example more clear, but we need to emphasize that you should not do this.

To compile the programs, type:

```
% cc xdrsample_server.c -o xdrsample_server
% cc xdrsample_client.c -o xdrsample_client
```

To execute the programs, first run the server on some host, preferably in background:

```
% xdrsample_server &
```

Then run the client, with three parameters: the name of the host on which the server is running, an integer, and a character. The client passes the integer and the character to the server, and displays the results. For example, if you run the server on a machine named terrapin and you type:

```
% xdrsample_client terrapin 4 w
```

Then you would see:

```
% Result = 5   x
```

A.10 Lisp

The following code uses Lisp to print out "Hello World." Because Lisp is interpreted, all you have to do is simply type:

```
(defun hello-world ()
    (format t "Hello World ~%"))
```

This defines the function to the Lisp interpreter. To evaluate the function (i.e., run the program) type:

```
(hello-world)
```

For complete details on Sun's Lisp implementation, please refer to the *Sun Common LISP User's Guide* and the *Sun Common Lisp Reference Manual.*

A.11 SunView

```
     /*****************
      * sunview_hello.c *
      *****************/

#include <suntool/sunview.h>  /* Sunview header file */
#include <suntool/canvas.h>   /* Canvas header file */

Frame my_frame;
Canvas my_canvas;
Pixwin *my_pw;

main()
{
     my_frame = window_create(NULL, FRAME,
          FRAME_LABEL, "Hello, world.",
          0);

     my_canvas = window_create(my_frame, CANVAS, 0);
     my_pw     = canvas_pixwin(my_canvas);
     pw_vector(my_pw, 0, 0, 200, 200, PIX_SET, 0);

     window_main_loop(my_frame);
}
```

This simple SunView program creates a frame, puts a canvas in the frame, opens a pixwin on the canvas, and draws a vector.

To compile it, type:

```
% cc sv_hello.c -o sv_hello -lsuntool -lsunwindow -
lpixrect
```

You must be in the SunView window system to execute sunview_hello. The tool that sunview_hello creates stays on your screen until it is destroyed, which can be done in one of two ways. First, with the mouse in the shelltool window from which you invoked sunview_hello, type Control-C. Or, you can point to the tool's black border with the mouse, hold down the right button to bring up a menu, and select Quit.

A.12 NeWS

```
/************
 * hello.ps *
 ************/

/hello_world {% x,y => - ( print "Hello World" at x,y)
     moveto
     /Times-Roman findfont
     15 scalefont
     setfont
     (Hello World) show
     showpage
} def

/makewin {% - => - (builds a test window)
     /win framebuffer /new DefaultWindow send def%
cliche
     {    % draw after damage
          /PaintClient {10 10 hello_world} def
          /FrameLabel (NeWS Demo) def% label window
     } win send         % execute contents of
                        %  {} in win's context

     /reshapefromuser win send % select size/position
     /map win send            % make visible
} def

makewin                 % run the program
```

This is a sample NeWS program that draws a window on the screen and writes “Hello World” in the window. Since the PostScript is interpreted rather than compiled, it is not necessary to compile the program.

In a terminal emulator window in the NeWS environment, type:

```
% psh hello.ps
```

For more information on NeWS, please refer to the *Sun NeWS 1.1 Manual*. To learn about PostScript, the two standard texts are the *PostScript Language Reference Manual* and the *PostScript Language Tutorial and Cookbook*, both written by Adobe Systems and published by Addison-Wesley.

A.13 SunCore

```
/****************
 * core_glass.c *
 ****************/

#include <usercore.h>

static float glassdx[] = {-10.0, 9.0, 0.0, -14.0, 30.0,
     -14.0, 0.0, 9.0, -10.0};
static float glassdy[] = {0, 1, 19, 15, 0, -15,
     -19, -1, 0};
int pixwindd();
struct vwsurf vwsurf = DEFAULT_VWSURF(pixwindd);

main()
{
     initialize_core(BASIC, NOINPUT, TWOD);
     initialize_view_surface(&vwsurf, FALSE);
     select_view_surface(&vwsurf);
     set_viewport_2(0.125, 0.875, 0.125, 0.75);
     set_window(-50.0, 50.0, -10.0, 80.0);
     create_temporary_segment();
     move_abs_2(0.0, 0.0);
     polyline_rel_2(glassdx, glassdy, 9);
     move_rel_2(-12.0, 33.0);
     line_rel_2(24.0, 0.0);
     close_temporary_segment();

     sleep(10);
     deselect_view_surface(&vwsurf);
     terminate_core();
}
```

This sample program above shows how to use SunCore in a SunView tool. The program creates a tool with a control panel and a canvas subwindow. Pushing the Draw button in the panel causes Core to draw a martini in the canvas.

The tricky part about getting SunCore and SunView to cooperate is telling the window system about where Core should draw. The window system expects programs to have an environment variable, called *WINDOW_GFX*, which contains the name of the window device upon which graphics are to be drawn. (To see this for yourself, type `printenv WINDOW_GFX` in a tty window; the response will be something like /dev/win18.)

Three lines in the sample program set the *WINDOW_GFX* environment variable to the appropriate value. These lines are:

```
canvasfd = (int)window_get(canvas, WIN_FD, 0);
win_fdtoname(canvasfd, window_dev_name);
we_setgfxwindow(window_dev_name);
```

The first line uses window_get() to get the file descriptor that corresponds to the canvas. The second line uses win_fdtoname() to convert the file descriptor (which is an integer, e.g. 18) to a device name string (e.g. /dev/win18). The third line sets the *WINDOW_GFX* environment variable to the device name string, using the we_setgfxwindow() function.

To compile the program, type:

```
% cc core_glass.c -o core_glass -lcore -lsunwindow -
lpixrect -lm
```

To execute, type `core_glass` in a terminal-emulator or graphics window, in the SunView environment.

A.14 SunCGI

```
/****************
 * cgi_sample.c *
 ****************/
#include <suntool/sunview.h> /* Sunview header file */
#include <suntool/panel.h>
#include <suntool/canvas.h> /* Canvas header file */
#include <cgipw.h>

Frame frame;   /* SunView frame */
Panel panel;   /* Control Panel */
Panel_item draw_pi;
Canvas canvas;/* SunView canvas */
Ccgiwin vpw;/* Window struct for CGI */

int drawbox_notify();
```

```
main()
{
     int name;

     frame = window_create(NULL, FRAME, 0);
     panel  =  window_create(frame,  PANEL,  WIN_HEIGHT,
60, 0);
     draw_pi = panel_create_item(panel, PANEL_BUTTON,
          PANEL_LABEL_IMAGE,
               panel_button_image(panel, "Draw", 0, 0),
          PANEL_NOTIFY_PROC, drawbox_notify,
          0);
     canvas = window_create(frame, CANVAS,0);

     open_pw_cgi();
     open_cgi_canvas(canvas, &vpw, &name);

     window_main_loop(frame);
}

drawbox_notify()
{
     Ccoor lr, ul;
     Ccoorlist coorlist;

     ul.x = 100;
     ul.y = 100;
     lr.x = 150;
     lr.y = 150;
     cgipw_rectangle(&vpw, &lr, &ul);
}
```

This is a sample program of CGI running in a tool.

For more information on SunCGI, please refer to the Appendixes D and E of *SunCGI Reference Manual*. It contains some more sample programs, and discusses using SunCGI in windows.

To compile the program, type:

```
%  cc  cgi.c  -o  cgi  -lcgi  -lsuntool  -lsunwindow  -
lpixrect -lm
```

To run, type `cgi_sample` into a terminal window while in the SunView window system. A new tool is created, with a panel at the top and a canvas at the bottom.

When you activate the Draw button in the panel (by pointing to it with the mouse cursor and clicking the left mouse button), CGI will draw a square in the canvas.

A.15 SunGKS

```
/****************
 * gks_sample.c  *
 *****************/

#include <gks/ansicgks.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/panel.h>

Gwpoint square[5] = { 0.2,0.2 ,
                      0.2,0.3 ,
                      0.3,0.3 ,
                      0.3,0.2 ,
                      0.2,0.2 };
Frame frame;
Canvas canvas;
Panel panel;
Panel_item draw_box_pi;

int draw_box();

main()
{
     Gws ws = 1;
     Gks_ws_config config;

     frame = window_create(NULL, FRAME, 0);
     panel = window_create(frame, PANEL,
          WIN_WIDTH,              500,
          WIN_HEIGHT,             28,
          WIN_X,                  0,
          WIN_Y,                  0,
          0);

     canvas = window_create(frame, CANVAS,
          CANVAS_RETAINED,        FALSE,
          WIN_WIDTH,              500,
          WIN_HEIGHT,             462,
          WIN_X,                  0,
          WIN_Y,                  33,
```

```
          WIN_BELOW,              panel,
          0);

     window_fit(frame);
     draw_box_pi   =   panel_create_item(panel,   PAN-
EL_BUTTON,
          PANEL_LABEL_IMAGE,panel_button_image(panel,
          "Draw a box", 0,0),
          PANEL_NOTIFY_PROC,      draw_box,
          0);

     if (gopengks(stdout, GMEMORY))
          exit(2);
     config = gks_ws_config_get_installed("sun_canvas");
     gks_ws_config_set(config,
          GKS_GP_USAGE,           GKS_USE_NONE,
          GKS_TEXTSW,             GKS_CREATE,
          0);
     gks_ws_config_install(config);
     gks_ws_config_destroy(config);
     gopenws(ws, canvas, "sun_canvas");
     gactivatews(ws);

     window_main_loop(frame);

     gdeactivatews(ws);
     gclosews(ws);
     gclosegks();
}
draw_box()
{
     Gseg seg = 1;

     gcreateseg(seg);
     gpolyline(5, square);
     gcloseseg();
}
```

This program illustrates GKS running in a SunView tool.

To compile the program, type:

```
% cc -o gks_sample gks_sample.c
     -lgks -lsuntool -lsunwindow -lpixrect -lm
```

All SunGKS programs expect the environment variable GKS-DIR to indicate the location of the GKS library subdirectory.

For example, if your GKS software is in a directory called /usr/unbundled/gks2.1, then the GKSDIR variable should be set to /usr/unbundled/gks2.1/lib/gks. To verify this, type:

```
% printenv GKSDIR
```

The response should be (in this example):

```
GKSDIR=/usr/unbundled/gks2.1/lib/gks
```

If the response from the system is incorrect or blank, you need to set the environment variable by typing:

```
% setenv GKSDIR /usr/unbundled/gks2.1/lib/gks
```

You can do this as needed, or you can edit the command into your .login file so that it will be executed automatically whenever you sign onto your workstation. For more information on GKSDIR, please refer to the *SunGKS Reference Manual*.

Once the environment is correct, you can run the program by typing:

```
% gks_sample
```

into any terminal-emulator window. The tool has two subwindows: a control panel and a canvas. Pushing the single button in the panel (its label says “Draw a box”) causes GKS to draw a box in the canvas.

For more example programs, please refer to Appendix G of the *SunGKS Reference Manual*.

A.16 SunPHIGS

```
/*****************
 * phigs_hello.c *
 *****************

#include <phigs/phigs.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/panel.h>

#define  WS    1
#define  ST    1

int go_notify();

main()
{
     Pwstype canvaswst;
```

```
     Frame frame;
     Canvas canvas;
     Panel panel;
     Panel_item go_pi;

     frame = window_create(NULL, FRAME, 0);
     panel  =  window_create(frame,  PANEL,  WIN_HEIGHT,
100, 0);
     go_pi = panel_create_item(panel, PANEL_BUTTON,
          PANEL_LABEL_IMAGE,
          panel_button_image(panel,"GO",0,0),
          PANEL_NOTIFY_PROC, go_notify,
          0);
     canvas=window_create(frame,   CANVAS,   WIN_HEIGHT,
600, 0);
     popenphigs((Pchar *)NULL, PDEFAULT_MEM_SIZE);
     canvaswst                                        =
phigs_ws_type_create(phigs_ws_type_sun_canvas, 0);
     popenws(WS, (Pconnid)canvas, canvaswst);

     window_main_loop(frame);

     pclosews(WS);
     pclosephigs();
}
go_notify()
{
     static Ppoint textpts = {0.1, 0.5};

     popenstruct(psetcharheight(.02);
     ptext(&textpts, "Hello world.");
     ppoststruct(WS, ST, 0.);
     pclosestruct();
}
```

This program shows how to use SunPHIGS to write some text in a SunView canvas. A frame is created with a control panel and a canvas. The canvas is made to be a PHIGS workstation, and window_main_loop() is executed. When the GO pushbutton in the panel is activated, a struct is opened, text is written in it, it is posted to the workstation, and then it is closed.

To compile the program, type:

```
% cc phigs_hello.c -o phigs_hello -llphigs -lphigs -
lsuntool -lsunwindow -lpixrect -lm
```

To execute the program, type:

```
% phigs_hello
```

into a terminal-emulator window within the SunView window system.

The SunPHIGS distribution tape contains a wealth of sample code in /usr/lib/phigs1.0/examples. Sources for these programs can be found in the appendixes of Getting Started with SunPHIGS. For further information on SunPHIGS, please refer to the SunPHIGS Reference Manual and the SunPHIGS Software Installation Guide.