1
Writing a Graphics Device Function for S

S graphics functions, (`plot`, `identify`, etc.) are device independent. They manage to produce output on a particular device through a few graphics primitives. The program that supplies these primitives for a particular device is known as a *graphics device function*.

S is distributed with a variety of graphics device functions. These functions allow S to perform graphical operations on AT&T 630 multi-tasking graphics terminals, Hewlett-Packard terminals and plotters, Tektronix terminals, printers that understand the PostScript page description language, the X and NeWS window systems, and printer-like devices. Nevertheless, new graphical devices (and libraries that perform graphics) are continually being developed. This chapter provides the information you will need in order to write your own graphics device function and install it into S.

1.1 A Simple Device Function

The production of a simple driver is straightforward; customizing the driver so that it makes optimum use of a particular device may require a substantial effort as well as a deep appreciation for the capabilities of the device and the way that S interacts with graphics devices. It's often a good idea to start from an existing function for a similar device.

Let's begin with something simple. Provided with the S source code is a file[†] `$H/src/qpe/skeleton.c` that contains a "working" device driver.

[†]File names are given with a shell variable `$H` that refers to the S home directory.

Follow these steps to try it out.

1. Find out where the S home directory is by typing the following line in response to the shell prompt `$`:

   ```
   $ H=`S SHOME`
   ```

   (These are back-quote characters, not apostrophes). This command stores the name of the S home directory in the shell variable `H`.
2. Make yourself a copy of the C program in your home directory:

   ```
   $ cd
   $ cp $H/src/qpe/skeleton.c .
   ```

3. On some machines, (DEC VAX running Ultrix or 4.3BSD and Sun 3 computers) you can dynamically load object code into a running version of S by using the `dyn.load` function. If you are using one of those machines, execute:

   ```
   $ cc -c -I$H/newfun/include -I$H/src/qpe skeleton.c
   $ S
   > dyn.load('skeleton.o')
   ```

   If your machine does not support dynamic loading, you will have to create your own version of S that contains the object code from `skeleton.c`:

   ```
   $ S LOAD skeleton.c
   $ S
   ```

   In either case, you are now running S with the skeleton object code available.
4. Type in the following S function:

   ```
   > simple <- function(ask = F, file = "") {
   +  graphics.off()
   +  z <- .C("simple",
   +      as.logical(ask),
   +      as.character(file))
   +  Device.Default("simple")
   + }
   ```

   The function `simple` is your new device function.
5. Now, you can execute `simple` and S will treat it as its current graphics device. Any graphical functions that are executed in S while the `simple` device is in effect will cause low-level graphical actions to be printed on your terminal. For example:

   ```
   > simple()
   Initialization String
   > plot(1:10)
   Clear the Screen
   ```

```
New Color 0
New Line Type 0
Move to 115, 132
New Rotation angle 0
New Size 1
Character '*'
Move to 204, 215
Character '*'
...
Flush
Move to 60, 216
New Rotation angle 90
Character '2'
...
Move to 86, 914
Line To 86,107
Line To 956,107
Line To 956,914
Line To 86,914
Flush
> q()
Wrap up execution
```

You can see from the printed output that the device function carries out simple actions such as:

- moving to a position in integer device coordinates;
- drawing a line from the current position to another position;
- plotting a single character at the current position;
- clearing the screen;
- setting simple graphical parameters such as color, line type, rotation angle;
- initialization, wrapup, and flushing of output.

Should you decide to write your own device function, your job will be to supply these actions.

Flush with success in getting this device function to work, perhaps you are now inspired to write one of your own. Section 1.2 tells how device functions fit into S. Section 1.3 describes how to write a device function for a low-level device by modifying the `skeleton.c` routine. Section 1.4 then describes device functions for somewhat smarter devices. Another file, `skeleton2.c`, is useful as a starting point for these higher-level device functions. Some hints on how to customize your device driver are presented in Section 1.5, followed in Section 1.6 by instructions on how to install your new driver into S for everyone to use.

1.2 How a Device Function Works

As you saw when trying the `simple` device function of section 1.1, an S device function is implemented as a function, written in the S language, that then calls a C language routine. This C routine initializes itself as a device function by creating a data structure that contains information about the device, including its current state and a vector of functions that the device provides to carry out operations. This data structure is presented to S via calls to the C routines `new_device` and `set_device`. Once these calls have taken place, S knows that this is now its current device for doing any graphics operations. >From this point on, any S functions that attempt to do graphics will eventually result in a call to one of the device's vector of functions.†

One of the first decisions you should make when writing a new device function is at which level the driver should operate. There are two popular models, representing two different types of graphics devices. The low-level model is for a device that doesn't know how to do very much graphically. We saw this level in our `simple` function. In this low level model, the device functions operate in a coordinate system preferred by the device, known as *device* or *raster* coordinates. The operations are typically simple: move to a specified point, draw a line from the current point to a specified point, draw a single character, clear the screen. This type of device is typified by a pen plotter or dumb (graphics) terminal.

The other popular graphics model is for higher level devices. They typically have mechanisms for translating from the *user's* coordinate system into the device coordinates and the device function implements operations such as: draw this set of lines, plot this text string at a specified point, draw a character or marker symbol at a list of coordinates. The high-level device is typically a graphics subroutine library, window system, or programmable graphics device. Usually, it is easier to write a low-level device function before attempting a high-level one.

†Part of the reason for this structure is that S may eventually have a wider range of graphics operations available. For example, this model allows S to have several graphics devices active at once; this would allow a plot to simultaneously draw on a terminal and write a file of PostScript description commands. Also, since each instance of a device contains its entire state, the same device could be active more than once, thus leading to several active graphics windows on the same screen.

1.3 Writing a Low-Level Device Function

Suppose that you have just bought the new Framus-3 plotter and are anxious to use it with S. Since the Framus-3 is a fairly simple-minded device, you choose the file `skeleton.c` as the basis for writing the device function. Make yourself a copy of `skeleton.c`

```
$ cp $H/src/qpe/skeleton.c $HOME/framus3.c
```

You can then modify the file `framus3.c` to create the C language code for your own device function.

Of course, you will also need an S function to invoke your C code; this should be adequate for now:

```
framus3 <- function(ask = F, file = "") {
    graphics.off()
    z <- .C("framus3",
        as.logical(ask),
        as.character(file))
    Device.Default("framus3")
}
```

The call to `graphics.off` ensures that any previously active device is terminated before the `framus3` device is activated; `Device.Default` takes care of initializing a few graphical parameters for the new device and also saves the name of the current device in frame 0 so that any S function can find out the name of the current graphics device.

Even though you will be modifying the file `framus3.c`, various places in this section will refer to the line-numbered listing of the file `skeleton.c` in Appendix 1. If you glance at Appendix 1, you will see several references to numbered graphical parameters, for example `am(15)`. This is a holdover from an early implementation where the parameters that described the device were numbered rather than given names. One advantage of the numbering scheme is that all of the parameters look alike (they are `am(xx)`), but we could have easily disguised the parameters by creating an include file with names rather than numbers. We didn't, however, except for a few parameters defined in lines 22–25. Console yourself that the device function is the only place where you ever need to know about the graphical parameter numbers, and even then only briefly. You can always make your own include file or put symbolic constants in your code if you wish.

1.3.1 *Initialization*

The first thing you should do in writing your device function is to get a reference manual for the Framus-3. The manual should tell you precisely what is necessary in order to get the device to do specific graphics operations. Look in the manual to find out the following basic characteristics of the device, and install the values in graphical parameters `am(20)` through `am(29)` as given in lines 108–117 of `skeleton.c`. You will need to know the *x*- and *y*-size of a raster (for example .01 inches), the minimum and maximum raster addresses for the device, and the size of a standard-sized character in rasters.

If you know the size of the device but do not know how big a raster is, you might compute it as follows:

```
am(28) = device_x_size_inches / (am(23) - am(22));
am(29) = device_y_size_inches / (am(25) - am(24));
```

If you know that the device has square rasters (identical physical dimensions in the horizontal and vertical directions), replace the second line with

```
am(29) = am(28);   /* rasters are square */
```

Parameters `am(20)` and `am(21)` give the size of a standard character, including surrounding space, in rasters. If there is no character size that is "standard", then you will need to pick a size that you want to consider the standard. This will be the default size that will appear on the plot when graphical parameter `cex` is `1`. When setting these parameters, remember to add on the inter-character space to `am(20)` and the inter-line spacing size to `am(21)`.

The constants in `am(26)` and `am(27)` tell where a character is addressed by the device, that is, if you move to a specific position and draw a character, where is the character located with respect to that position? For example, the values `(0,0)` indicates that the lower-left corner of the character is the addressed position; `(.5,.5)` indicates that the character is addressed at the center, etc. Sometimes the correct values of these parameters must be determined experimentally; try plotting a large character atop a pair of crossed lines to see if the centering is correct.

The value in `am(31)` (line 119) should be `0` if the device cannot rotate characters (rare these days) and `1` if it can. The value in `am(1)` (line 120) should be zero if the device cannot change character size (again, increasingly rare) and non-zero if it can. Both of these parameters are only rough indicators of the capabilities of the device. Setting them to zero means that S will never try to use the parameters, while non-zero values mean that the size or rotation capabilities are continuously variable. There is no way to indicate that a device only has a selected set of allowable character sizes or rotations.

Once upon a time, the graphics software underneath S wanted to be able to tell which device was active. To do so, it required that `am(30)` be set to an arbitrary unique negative number for each different device.[†] If you want to be tidy about it, make up a negative number and include it in the file

[†]What happened to the positive numbers? They were once supplied by a graphics subroutine library that is no longer in use.

`$QPE/device.h` as a defined constant, different from the other device numbers that appear there.

Installing the correct parameters in lines 108–120 is actually a fairly difficult part of the overall task of writing a device function, but we'll assume you have accomplished this task. Let's now discuss the routine `simple` in lines 94–139. Change its name in line 95 to `framus3` to reflect the name of your own graphics device. This is the only externally visible entry point for the device function; notice that all other routines in the file are declared `static`. As it is currently written, routine `framus3` takes two arguments: a logical vector `a_ask` and a character vector `a_file`. You can make your function take any set of arguments that will help the user control the device. It is a convention, however, that most devices take arguments allowing the user to say "ask me before you advance to the next page" and "put all of the output you would have sent to the device into a file instead." The value of `a_ask` is simply saved away in a local variable in line 122 (it will be referred to later by the `clear` routine that advances to a new page).

If you refer to line 14, you will see that the local variable `ask` is actually one of a vector of local state variables kept by the device. The rationale for these state variables is that, if all local variables are held in the device structure, there will be nothing to prevent S from having multiple copies of the device function active at the same time. Although this capability is not presently implemented, we anticipate future versions of S that will support multiple graphs in different windows on a multi-window device.

The `a_file` argument is taken care of in lines 124–129. The `framus3` routine looks to see if the `a_file` argument was supplied, and if so, it tries to open the specified file for appending. If it cannot open the file, it calls routine `Recover` with an error message. In any case, by the time we reach line 131, the file pointer `outfile` has been set to point to the file where all output should go.

Now is as good a time as any to describe the data structure that is initialized by `framus3`. Look at the declaration for the static data structure `device` that occupies lines 38–75. Lines 39–44 are place holders that will be filled in by the routine `new_device` called in line 103. You should leave them alone. Lines 47–73 are an ordered list of function names. The only one you will need to modify is line 47; change `simple` to `framus3`. This list gives the routines that perform the actions required of a device driver. S knows that the 5th function in the list is always the routine that will cause this device to plot a set of points. The remainder of this section will describe these routines in detail.

1.3.2 *Wrap Up:* `wrap()`

The `wrap` routine is called when the device is used for the last time. This is a good place to put away pens, close files, erase graphics windows, etc. On some devices, no wrapup action will be necessary.

1.3.3 *Flush:* `flush()`

Some devices have a distinction between *alphanumeric* mode and *graphic* mode. When the device is in alphanumeric mode, characters received are displayed normally; when it is in graphic mode, characters received are treated as graphics instructions and operands. Other devices (a pen plotter attached to a terminal) may need to be turned on and off. It is the responsibility of the `flush` to handle the *turn-off* side of these situations. It is called whenever it is possible that non-graphics output will be sent to the device. (The `seek` routine, described later, handles the *turn-on* operation.) Of course, if the graphic device is not an interactive terminal, there is no need for the `flush` routine to do anything.

Some care should be taken to make the alphanumeric/graph mode switching reasonably efficient. The graphical subroutines underlying S call the `flush` routine whenever it is possible that there might be alphanumeric output. This may cause lots of mode switching. Remember, your routine can probably ignore mode switching if it is directing the output to a file.

1.3.4 *Signal Catching:* `signalled()`

The `signalled` routine is called by S whenever the user hits an interrupt while the device function is in control, or more generally, if a signal is generated when the device function is executing. Such signals may eventually cause error messages to be printed or may just terminate execution of the current S expression. In many cases, the `signalled` routine need do nothing. However, for some devices, particularly if the device has alphanumeric and graphic modes, the `signalled` routine should be written to get the device back into alphanumeric mode. It may also need to send instructions that can terminate a partially completed graphic operation (for example, imagine that the user interrupts output in the middle of sending a polygon description) prior to exiting graphic mode. Another use for `signalled` is to clean up allocated storage that may have been left around due to graphics operations.

1.3.5 *Points, Lines, etc.*

The device functions specified in lines 57–61 and 63 are generic routines that need no modification for a low-level device driver. These routines turn high-level device actions into low-level device actions. For example, `bpntsz` takes a vector of user coordinates for a set of points to be plotted, turns those coordinates into device coordinates and repeatedly calls the `seek` and `point` routines to plot the points. Section 1.4 describes how to write these routines for high-level device functions.

1.3.6 *Clear the Screen:* `clear()`

The `clear` function is fairly straightforward. It should interrogate the user if requested (remember the `ask` argument to the initialization routine). This gives the user some time to look at the present display before it is erased. `Clear` should then clear the screen of a terminal, insert a new sheet of paper into a pen plotter, or ask the user to perform the action if there is no automatic way to handle it.

1.3.7 *Seek:* `seek(x,y)`

The `seek` operation is perhaps the most fundamental graphic operation for a low-level graphics device. The idea is that `seek` will cause the device to be positioned (physically or logically) at the specified device coordinates. For some devices, `seek` will cause the device to do something, e.g., a pen plotter will move to the specified `x,y` device coordinates with the pen up. On other devices, `seek` will do nothing more than store the specified `x,y` coordinates for later use.

On all devices, `seek` is called prior to any action that will actually cause any output to appear on the device, and thus is the standard place to insert actions that are required before doing graphics. As explained in the section on `flush`, `seek` is the place to insert code that will turn a graphic device from alphanumeric mode into graphic mode.

It is also convenient to use `seek` as a place to check the graphical parameters for line type, color, and line width, to see if they have changed. (It is better to test these line-oriented parameters in `seek` than in `line` because a connected sequence of lines is drawn by means of a single `seek` followed by a number of calls to `line`. The software will never change graphical parameters in the middle of such a sequence, so it is unnecessary to check changes in line-oriented parameters for each call to `line`.)

1.3.8 *Point:* `point(ich,crot)`

The function of the `point` routine is to put out a single character at the current location. If the device can rotate characters (determined by `am(31)` in the initialization routine), then it is up to `point` to carry out the rotation. The rotation angle is in degrees, measured counterclockwise from the horizontal.

When S puts out an entire text string, it does so by repeated calls to `seek` and `point`. The higher-level software (actually `btextz` as specified in line 60) takes care of computing inter-character spacing and string rotation. (You supplied the character size in lines 108–109.) In order to handle devices that are capable of proportionally-spaced text, where the width of each character is variable, you must implement the higher-level `text` function (and you probably still need to implement `point`, since it is called by the default `bpntsz` function).

Because it is called for each character, `point` is an appropriate place to check for changes in character size, although `seek` would be another place that such checking could be carried out.

1.3.9 *Line:* `line(x,y)`

This function is easy—it simply draws a line from the current location to the specified `x,y` coordinate position (in device coordinates). After drawing the line, the device is assumed to have `x,y` as its new current position. Generally, all checking for changed graphical parameters that might affect the line should already have occurred in the `seek` routine.

1.3.10 *Graphic Input:* `getxy(x,y,flag)`

The `getxy` function is unlike the other device functions in that it returns, in its arguments, information from the device. As such, it is only useful in interactive situations. If the device provides no explicit means of entering coordinate positions, you can use the routine `bquxyz` to implement `getxy`; it will prompt the user to type in the coordinates at the terminal.

Otherwise, the `getxy` routine should prompt the user to select a position on the screen (by the mouse, by cursor keys, or by adjusting cursor-control knobs) and give some indication that the position has been selected (by clicking a mouse button or typing a character on the keyboard). The routine should set the `x,y` arguments to reflect the device coordinates of the position and also return a value `flag` that tells whether the input was successful. Unsuccessful input, where the user decided not to enter a point, is indicated by a negative value in `flag`. (Normally, the user indicates a desire to not input another point by typing a carriage return or by clicking two mouse buttons simultaneously, or by whatever signal you, as author of the device function, deem appropriate.) For successful input, `flag` usually contains an indication of which character the user typed, which button was hit, etc.

The `getxy` function may be the most difficult function for you to write. It may require you to use routines `echo_off` and `echo_on` to control echoing. You may also have to disable interrupts during the time that echoing is turned off so as not to leave the terminal in a non-echoing state if an interrupt does occur. For an example, look at the `getxy` routine for the S device function `tty5620` located in file `$H/src/qpe/Stty5620.c`.

1.3.11 *Menu:* `menu(ent,arglist)`

The `menu` routine, if provided, gives a graphical way of displaying a set of character strings to the user and getting back a number representing the selection of one of those choices. Unless your device has an especially good way of presenting a menu, it is probably best to not implement the `menu` routine; by default, menus will be presented on the user's terminal.

If you want to implement your own menu function, take a look at the `mymenu` routine in Appendix 2, lines 160–179.

1.3.12 *Hook:* `hook(type,x,n,y,m)`

Caution: the material in this section is likely to undergo major revisions in future releases of S. Try to avoid `hook` *if possible.*

The set of operations provided for each device can be extended in a device-dependent way by means of the `hook` routine. Hooks can be used to set the color table, to access circle or arc generators, etc. The `hook` routine is called with a operation number in argument `type`, along with a vector of `n` arguments to the operation (stored in `x`). The operation can also return a vector of output values (in `y`, with its length in `m`). That means that the hook action can be used for output or input from the graphics device.

There are a few pre-defined hook operations named in the `device.h` include file. These include `HOOK_setcolor` whose purpose is to set the color lookup table. Lines 250–268 of `skeleton.c` show a sample hook function.

To invoke a hook operation from S, call the function `hook`:

```
set.color <- function(defs) {
    if(!is.matrix(defs) || ncol(defs)!=4)
        stop("Input is not a color table")
    invisible(hook(1, defs))
}
```

The routine `set.color` takes a 4-column matrix as an argument and makes color table settings. The constant `1` passed to `hook` is the numeric value corresponding to operation type `HOOK_setcolor`. Since operation `HOOK_setcolor` produces no output, we wrap the call to `hook` with function `invisible` to prevent automatic printing of the `numeric(0)` value that is always returned by `set.color`. Had our hook routine returned output values in `y`, they would be the vector returned by the S `hook` function.

1.4 A High-Level Device Function

This section describes how to write a high-level device function. Appendix 2 provides a listing of `skeleton2.c` which is a sample implementation of such a device function.

A high-level device function differs from a low-level function in that its plotting routines take vectors of arguments in the user's coordinate system. It is the responsibility of these routines to produce an entire collection of marks on the page (set of points, connected line segments), to map the user coordinates into the device coordinate system, and to clip any objects that are totally or partially outside of the plotting area.

You might wonder why anyone would want to write high-level device routines with all of these requirements when low-level routines are so easy to write. The reason is that there are many graphic "devices" that are more naturally approached at a high level. Consider, for example, a GKS graphics library as a family of devices. The GKS library itself may well provide access to a number of different pieces of graphics hardware, and the routines in the library are precisely the high-level kind—they produce collections of points or lines as well as handle coordinate transformations and clipping. Another example of a high-level device might be a "smart" terminal, perhaps running the X window system.

The next few sections will describe the routines needed by high-level device functions. The low level `seek`, `point`, `line`, and `getxy` routines may not be needed for a high-level device function.†

1.4.1 *Points, Lines, and Segments:* `points(x,y,n)`

We will use `points` as an example high-level routine. The routines `lines` and `segments` are directly analogous.

The job of the `points` function is to display the character whose ASCII value is stored in `am(15)` (corresponding to graphical parameter `pch`) at specified positions on the device. One special concern of `points` is that there are several special values of `pch` that need to be taken into consideration.

- Values of `pch` that are less than 32 correspond to non-printing ASCII characters. S uses these values to index one of a set of vector-drawn plotting symbols. The Fortran-callable routine `dmarkz` can be used to draw these symbols. If desired, plotting characters can also be defined

†If you do not supply a high level `text` routine, you will need to supply the low-level `seek` and `points` routines. If you do not supply a high level `points` routine, you will need to supply the low-level `seek` and `points` routines. If you do not supply a high level `lines` routine, you will need to supply the low-level `seek` and `line` routines. If you do not supply a high level `locate` routine, you will need to supply the low-level `getxy` routine.

for values of `pch` that are greater than 127, beyond the range of the ASCII character set.

- If `pch` is a period ("."), then `points` should draw a small plotting dot at each position. The plotting dot can be an actual centered period, or can be made simply by turning on one or two rasters at the specified locations.
- Any other value of `pch` is interpreted as the integer value corresponding to an ASCII character; the associated character should be plotted (centered, if possible) at the specified locations.

There are other graphical parameters that should be taken into account by the `points` routine. They are character size (`am(18)`), color (`am(10)`), and perhaps font (`am(79)`). In addition, `lines` and `segments` should, if possible, also honor line width, (`am(9)`), and line type (`am(8)`).

Any of the high-level routines can make use of the macros `UxR` and `UyR` if it is necessary to convert from user coordinates into device or raster coordinates. Similarly, macros `RxU` and `RyU` convert from raster coordinates into user coordinates (useful for graphic input).

1.4.2 *Polygon:* `polygon(x,y,n)`

The `polygon` routine is similar to `points`, but instead of plotting a set of points, it uses those positions to define the vertices of a polygon which the device then causes to be filled in the current color. It is up to you to decide if the `polygon` routine should provide any other fill patterns, polygon outlines, etc. One possibility is to use negative color numbers to select a different style of filling. Be creative: it may be possible to encode dithering information in the color parameter, too.

1.4.3 *Text:* `text(x,y,buf,n,pos)`

The `text` routine is responsible for displaying a character string located relative to the specified `x, y` coordinates. The string itself is `n` characters long and is located in `buf`. Argument `pos` controls justification of the string: if `pos` is `0`, the string is left-adjusted at the specified coordinates; if `pos` is `1`, the string is right-justified; and if `pos` is `.5`, the string is centered. Alternatively, think of a position a fraction `pos` of the way from the beginning of the string to the end. That position is centered at the specified coordinates. It is best if `pos` is treated as a continuously varying parameter that may even go outside of the interval from `0` to `1`.

The nice thing about implementing a `text` function is that it can produce proportionally-spaced character strings. (Also, note that the character size specified in `am(20)` and `am(21)` is only a nominal size and not used in any critical computations.) It is up to the `text` routine to provide appropriate actions corresponding to graphical parameters `am(18)` (character size), `am(48)` (string rotation), `am(14)` (character rotation), and `am(10)` (color). The `text` routine may also want to take advantage of the `font` parameter stored as an integer in `am(79)`.

1.4.4 *String Length:* `slen(str,inches)`

At present, the string length function is not used, so you don't need to implement it. The purpose of the routine, though, is to allow the device to determine how long a character string will be (measured in physical units such as inches). The reason this is device dependent is that many of the smarter devices have their own proportionally spaced fonts and it may be difficult for anything but the device to determine the true physical length of any particular string.

1.4.5 *Locator:* `locate(x,y,n,nmax)`

The `locator` function allows the user to perform graphic input for a collection of points. It is most often implemented by the routine `brdpnz` which in turn repeatedly calls the low level `getxy` routine. On input, the 4th argument to `locator` gives the maximum number of points the user is allowed to input. It then repeatedly reads the graphic input device until this maximum is reached or until the user signals end of input. The user coordinates of the input points are returned in the first two `x, y` arguments, and the number of points is returned in the third argument.

1.5 What Next?

As we mentioned before, it is generally best to start by implementing a device function that is as simple as possible and later to add support for the various features of the device. This section will present some ideas for such enhancements. In many cases, examples will be given of S device functions with such enhancements. Look at the S source code for the functions to see details of the implementation.

Some devices, particularly those that are not inherently interactive, may be enhanced by the capability to pipe their output directly into a UNIX command, rather than storing it in a file. For example, the postscript device function, by default, pipes its output to a laser-printer spooling command. Similarly, the `pic` function can pipe into *troff* and `printer` can send its output to a line printer spooler.

Writing a device function for a window system is often a difficult task—

you must try to adhere to the philosophy of the window system as well as strike a good balance of simplicity, efficiency and use of special features. When writing such a device function, it is desirable to get advice from a programmer who is expert in the window system.

Many window-based terminals allow the user to reshape windows at will. The implications for a device function that runs in such a windowing environment are that the actual raster coordinates for the device may change during execution. What should the device function do in case of a reshape operation? One possibility is for the device to instantly react to such an operation, with all new graphics being targeted to the new window size and position and any graphics presently in the window recreated in the new one. Unfortunately, that is a difficult task, requiring the device function to keep track of all graphics commands that have been executed since the most recent `clear` operation. It is generally much simpler if the S device function does not itself have to keep a display list in order to recreate the plot in a reshaped window. It is probably sufficient if S provides the user with the capability of producing a new plot in the reshaped window.

In any case, following a window move or reshape operation, the device function should call the routine `zzdevz` and give it the new limits for the device coordinates (see `src/qpe/Stty5620.c` for an example). This routine recomputes the graphical parameters used for the user- to device-coordinate mappings as well as device size, etc.

Some issues in writing a device function for a specific piece of hardware are difficult to resolve. For example, how is it best to use the device for graphics? In a multi-window terminal, for example, it is often good to send the graphics to a window that is disjoint from the window where the user types S expressions. That enables the windows to be sized and positioned to reflect the specific requirements of the problem.

There are times when certain devices are most accurately modeled by mixtures of high and low level routines. For example, the `pic` function uses primarily low-level routines, but it implements its own high-level `text` function so that it can make use of proportionally spaced text strings.

There are several portability issues involved in writing device functions. First, the macro `F77_SUB` should be used when defining a C routine that may be called by Fortran routines or when calling a Fortran routine from C. The macro is defined at S installation time to make the appropriate mapping from a C-based name to the corresponding Fortran name.† In general, all routines that are called by Fortran get pointers to their arguments. Similarly, whenever calling Fortran routines, pass pointers to arguments. The macros `F_CHARTYPE` and `F_CHARP`, used in device routines that receive character data, insulate the code from various ways that Fortran compilers pass character variables. The macros are defined in `newfun/include/cdefs.h` in a machine-dependent way. `F_CHARP` generates a pointer to the actual character string declared with the type `F_CHARTYPE`.

†For example, on most UNIX systems, the `f77` Fortran compiler adds a trailing underscore charac-

Although some graphical parameters have obvious interpretations, For the most part, the precise treatment of graphical parameters is determined by each device function. For example, the character expansion parameter (`cex`) imposes some constraints on the device function because characters of size 2 ought to be twice as big as characters of size 1. However, the physical size of a character of size 1 is up to you as you write the driver. In general, the size 1 characters should be those that are most natural for the device because it will be the frequently used default size.

Other parameters, such as color, line style and font, are generally interpreted as indices into a device-dependent list of suitable values. In the case of color, a device will often have a built-in color mapping capability, where the precise color is determined by a table that the device contains. As we saw earlier, one of the `hook` routines is often used to allow the user to control the contents of the color table. One general notion is that low-numbered colors should be visually distinct from one another, since it is convenient for a user who is plotting several sets of data to do them in colors 1, 2, ...

The parameter for line width (`lwd`) is very flexible. It can be treated as a continuous numeric value, so that line width 2 is twice as wide as the standard line width of 1. Alternatively, it can be implemented as a table of discrete widths.

It is usually good practice to allow all values of a particular parameter rather than to have the device function issue error messages. This can be done by cycling repeatedly through the table of allowed values or by mapping all out-of-range values into a specific value, etc.

1.6 Installing the Device Function

Now that you have written the spiffy new `framus3` device driver, you might want to install it into the default version of S at your installation. This is quite easy. First, move the file `framus3.c` into the `$QPE` directory. Add `framus3.o` to the `DEVICE_FILES` list at the top of `$QPE/Makefile.` Finally, add two lines describing `framus3` to `$QPE/S_ld_init.c`†. These lines will look like:

```
extern framus3();
{SYMBOL(framus3),(long)framus3,NULL},
```

Now you can remake the S executive by:

---

ter onto the name of any routine. The `F77_SUB` macro appends that underscore to make C routines callable by Fortran and to make Fortran routines accessible to C.

†In older versions of S this file was known as `S_load_init.c`.

```
$ cd $QPE
$ make
```

This will make a new S executive that you can test by executing

```
$ S nSqpe
```

When you have confirmed that this version is working, execute

```
$ mv $C/Sqpe $C/oSqpe  # save a backup copy
$ mv $C/nSqpe $C/Sqpe  # install the new one
```

Also, be sure to copy your S function `framus3` into directory `$SHOME/s/.Functions`, and add your documentation file (you did document your function, didn't you?) to directory `$SHOME/s/.Functions/.Help.` At this point, your new device function will be usable by everyone on your machine.

Appendix 1

```
/* S device driver for simple graphics devices
   (pen plotters, simple crt’s, etc.
*/

#include <stdio.h>
#include "cdefs.h"
#include "structure.h"
#include "device.h"

/* number of local state variables to be retained by device */
#define NLOCAL 5

/* definitions for the state variables */
#define ask ((long *)(cur_device->local_params))[0]
#define oldcolor ((long *)(cur_device->local_params))[1]
#define oldltype ((long *)(cur_device->local_params))[2]
#define oldrot ((long *)(cur_device->local_params))[3]
#define oldsize ((float *)(cur_device->local_params))[4]

/* defines that allow easy access of graphical parameters */
#define am(i)     (F77_COM(bgrp)[i-1])
#define Ltype     ((int)am(8))
#define Color     ((int)am(10))
#define Cex    (am(18))
#define notNew    (am(121))
extern float F77_COM(bgrp)[];

/* declarations for routines in the device structure
   note that simple() is the only externally-called function
   all the rest are statics appearing in the device structure */
vector *simple(),
   *F77_SUB(bpntsz)(), *F77_SUB(blinsz)(), *F77_SUB(bpolyz)(),
   *F77_SUB(btextz)(), *F77_SUB(bsegsz)(), *F77_SUB(brdpnz)();
static vector  *wrap(), *flush(), *signalled(),
   *clear(), *hook(), *seek(), *point(), *line(), *getxy();

/* structure that describes the device and its functions */
static device d_simple = {
         /* leave these alone */
   FALSE,         /* active flag */
   0,          /* index in list of devices */
   (float *)NULL, /* copy of parameter array */
   NLOCAL,        /* number of local parameters */
   (char *)NULL,  /* slot for the local parameters */
         /* functions performed by driver */
   {     /* <R> required <O> optional */
      simple,     /* initialize <R> */
```

```
      wrap,       /* wrap up <R> */
      flush,      /* flush <R> */
      signalled,  /* caught signal <R> */
         /* these functions deal with sequences of points,
            lines, segments, etc., and can be
            written specially for this device or
            can be the parenthesized pre-defined
            functions that call lower-level seek,
            line, character, etc functions */
      F77_SUB(bpntsz),/* points <R> (bpntsz) */
      F77_SUB(blinsz),/* lines <R> (blinsz) */
      F77_SUB(bpolyz),/* polygon <R> (bpolyz) */
      F77_SUB(btextz),/* text <R> (btextz) */
      F77_SUB(bsegsz),/* segments <R> (bsegsz) */
      clear,      /* clear <R> */
      F77_SUB(brdpnz),/* graphic input <R> (brdpnz) */
      NULL,       /* menu <O> */
      hook,       /* hook <O> */
         /* these low-level functions must be supplied only if
            bpntsz, blinsz, bsegsz, btextz, brdpnz are
            used above */
      seek,       /* seek <O> (low level) */
      point,      /* point <O> (low level) */
      line,       /* line <O> (low level) */
      NULL,       /* length of string <O> (ignore this) */
      getxy,      /* input <O> (low level) */
   }
};

FILE *outfile;

/* this is the device initialization function that is called
   from the device driver interpreted function using the .C function.
   For example, this function is called by

   simple <- function(ask = F, file = "")
   {
      graphics.off()
      z <- .C("simple",
         as.logical(ask),
         as.character(file))
      Device.Default("simple")
   }

*/

vector
*simple(a_ask,a_file)
long *a_ask;
char **a_file;
{
```

```
   device *d, *new_device();
   int i;

   /* initialize device structure */
   d = new_device(&d_simple, 0L);
   set_device(d->which);

   for(i=1; i<=39; i++)
      am(i) = 0.0;
   am(20) = 11.;  /* char size (x) in rasters */
   am(21) = 21.;  /* char size (y) in rasters */
   am(22) = 0.;   /* minimum x raster coordinate */
   am(23) = 1000.;/* maximum x raster coordinate */
   am(24) = 0.;   /* minimum y raster coordinate */
   am(25) = 1000.;/* maximum y raster coordinate */
   am(26) = 1./6.;/* char addressing offset x */
   am(27) = .25;  /* char addressing offset y */
   am(28) = .001; /* x raster size in inches */
   am(29) = .001; /* y raster size in inches */
   am(30) = -2000;/* arbitrary negative device number (device.h) */
   am(31) = 1; /* allow char rotation */
   am(1)  = 2.;   /* allow char size change */

   ask = *a_ask;  /* initialize local variables */

   if(**a_file){  /* no interaction allowed if output to file */
      outfile = fopen(*a_file,"a");
      if(outfile == NULL) Recover("Cannot open file", S_void);
      ask = 0;
      }
   else outfile = stdout;

   fprintf(outfile,"Initialization String\n");

   /* these values for the old parameter values ensure
      that the parameters will be set first time around */
   oldcolor = oldltype = oldrot = oldsize = -1;
   F77_SUB(defltz)();/* set default values for other parms */
   notNew = 1; /* force erase on first real plotting */
   return(S_void);
}

/* this function is called to wrap up execution */
static vector *
wrap()
{
   fprintf(outfile,"Wrap up execution\n");
   if(outfile != stdout) fclose(outfile);
   return(S_void);
}
```

```c

/* function executed when the device receives a signal (like an interrupt)
   its purpose is to make sure the device is reset to a consistent
   state when interrupted */
static vector *signalled()
{
   fprintf(outfile,"Signal Received\n");
   return(S_void);
}

/* draw a line segment */
static vector *line(x,y)
long *x,*y;
{
   fprintf(outfile,"Line To %ld,%ld\n",*x,*y);
   return(S_void);
}

/* put out a single character at current position */
static vector *point(ich,crot)
F_CHARTYPE ich; float *crot;
{
   /* change rotation angle if needed */
   if(*crot != oldrot){
      fprintf(outfile,"New Rotation angle %g\n",*crot);
      oldrot = *crot;
      }

   /* change character size if needed */
   if(Cex != oldsize){
      fprintf(outfile,"New Size %g\n",Cex);
      oldsize = Cex;
      }

   fprintf(outfile,"Character ’%c’\n",*F_CHARP(ich));
   return(S_void);
}

/* position the device at x,y coords
   this is also the place to check color and line style changes */

static vector *seek(x,y)
long *x,*y;
{
   if(Color != oldcolor){
      fprintf(outfile,"New Color %g\n",Color);
      oldcolor = Color;
      }

   if(oldltype != Ltype){
      fprintf(outfile,"New Line Type %g\n",Ltype);
```

```
        oldltype = Ltype;
        }

    fprintf(outfile,"Move to %ld, %ld\n",*x,*y);
    return(S_void);
}

/* clear the graphic area */
static vector *clear()
{
    if(ask && outfile==stdout){
        fprintf(stderr,"GO? ");
        fflush(stderr);
        while( getc(stdin)!='\n' ) ;/* ignore reply */
        }
    fprintf(outfile,"Clear the Screen\n");
    return(S_void);
}


/* called to make sure device is flushed and ready to accept
    text as well as graphic output */
static vector *flush()
{
    fprintf(outfile,"Flush\n");
    fflush(outfile);
    return(S_void);
}


/* read an x-y pair from the device, flag indicates whether
    read was successful or not (a negative value indicates
    no point given and generally terminates higher-level reads)
*/
static vector *getxy(x,y,flag)
long *x,*y,*flag;
{
    if(outfile!=stdout){
        /* non-interactive, should read from terminal */
        F77_SUB(bquxyz)(x,y,flag);
        return(S_void);
        }
    fprintf(outfile,"Ready to read x-y pair\n");
    /* the code here should fill up x and y and
        set flag to 0 for success, -1 for failure */
    *x = 100.; *y = 100.; *flag = 0;
    return(S_void);
}

static vector *hook(type, x, n, y, m)
long *type, *n, *m;
```

```
float x[], y[];
{
   switch((int)*type) {
   case HOOK_setcolor:
      {
      int ndef = *n / 4, i;
      float *N = x, *H = N+ndef, *L = H+ndef, *S = L+ndef;
      printf("New color definitions:\n");
      for(i = 0; i < ndef; i++)
         printf("\t%g: %g %g %g\n", *N++, *H++, *L++, *S++);
      break;
      }
   default:
      printf("unknown hook type: %d\n", *type);
   }
   return(S_void);
}
```

Appendix 2

```
/* device driver that uses high-level routines */

#include <stdio.h>
#include "cdefs.h"
#include "structure.h"
#include "device.h"

extern float F77_COM(bgrp)[];
#define am(i)     F77_COM(bgrp)[(i)-1]

/* note that all high-level device routines get user coordinates */
/* may need to transform them to rasters for plotting */
/* user-to-raster and raster-to-user conversions */
#define Xorigin   (am(36))
#define Yorigin   (am(38))
#define Xscale    (am(37))
#define Yscale    (am(39))
#define UxR(x)    ((int)((x) * Xscale + Xorigin))
#define UyR(y)    ((int)((y) * Yscale + Yorigin))
#define RxU(x)    (((x) - Xorigin) / Xscale)
#define RyU(y)    (((y) - Yorigin) / Yscale)

vector *highlevel(), *F77_SUB(brdpnz)(), *F77_SUB(bquxyz)();
static vector *wrap(), *flush(), *signalled(), *points(), *lines(),
      *polygon(), *text(), *segments(), *clear(), *mymenu();

static device d_highlevel = {
   FALSE, 0, NULL, 0, NULL,
   {highlevel, wrap, flush, signalled, points, lines,
    polygon, text, segments, clear, F77_SUB(brdpnz), mymenu,
    NULL, NULL, NULL, NULL, NULL, F77_SUB(bquxyz)}
};

vector
*highlevel()
{
   device *d, *new_device(); int i;

   /* initialize device structure and graphical parameters */
   set_device(new_device(&d_highlevel, 0L)->which);
   for(i = 1; i <= 39; i++)
      am(i) = 0;
   am(20) = 10;   /* character size */
   am(21) = am(20) * 1.2;
   am(22) = 0;    /* x limits */
   am(23) = 1000;
   am(24) = 0;    /* y limits */
```

```
   am(25) = 1000;
   am(28) = .001; /* raster size in inches */
   am(29) = am(28);
   am(30) = -9876;/* some magic negative number */
   am(31) = 1;    /* characters rotate */
   am(1) = 1;     /* characters scale */
   F77_SUB(defltz)();/* set other parameters */
   printf("Initialize\n");
}

/*
 * flush() brings the output up to date
 * clear() prints the current page
 * wrap() wraps up the job
 * signalled() fixes up if interrupt or other signal occurs
 */
static vector *
flush()
{
   printf("Flush\n");
   return(S_void);
}

static vector *
clear()
{
   printf("Clear\n");
   return(S_void);
}

static vector *
wrap()
{
   printf("Wrapup\n");
   return(S_void);
}

static vector *
signalled()
{
   printf("Signalled\n");
   return(S_void);
}

static vector *
lines(x, y, n)
float *x, *y;
long *n;
{
   int nn = *n;

```

```
   printf("Move to %d %d\n", UxR(*x++), UyR(*y++));
   while(--nn)
      printf("Line to %d %d\n", UxR(*x++), UyR(*y++));
   return(S_void);
}

static vector *
segments(x1, y1, x2, y2, n)
float *x1, *y1, *x2, *y2;
long *n;
{
   int nn = *n;

   while(nn--)
      printf("Segments %d %d %d %d S\n", UxR(*x1++), UyR(*y1++),
                   UxR(*x2++), UyR(*y2++));
   return(S_void);
}

static vector *
polygon(x, y, n)
float *x, *y;
long *n;
{
   int nn = *n;

   printf("Start Polygon %d %d\n", UxR(*x++), UyR(*y++));
   while(--nn)
      printf("Line to %d %d\n", UxR(*x++), UyR(*y++));
   printf("End Polygon\n");
   return(S_void);
}

static vector *
points(xx, yy, n)
float *xx, *yy;
long *n;
{
   long nn = *n, pch = (long)am(15); /* yuk */

   /* if plotting character is less than 32, then call special routine
      to draw plotting symbols rather than characters */
   if(pch < 32)
      F77_SUB(dmarkz)(xx, yy, n, &pch);
   else
      while(nn--)
         printf("Point %d %d\n", UxR(*xx++), UyR(*yy++));
   return(S_void);
}

static vector *
```

```
text(x, y, buf, n, pos)
float *x, *y, *pos;
F_CHARTYPE buf;
long *n;
{
   printf("Text ’%s’ at %d %d (adjust %g)\n", F_CHARP(buf),
      UxR(*x), UyR(*y),*pos);
   return(S_void);
}
#include "util.h"
static vector *
mymenu(ent,arglist)
vector *ent, *arglist;
{
   vector *value;
   int n, i;
   UNUSED(ent);
   value = coevec(*(arglist->value.tree),ANY,TRUE,CHECK_IT);
   if(value==NULL_ENTRY) return(S_void);
   value = coevec(value,CHAR,TRUE,CHECK_IT);

   printf("Menu with %d items:\n", value->length);
   for(i=0; i<value->length; i++)/* the labels */
      printf("%d: %s\n",i+1,value->value.Char[i]);
   scanf("%d",&n);
   while(getchar()!=’\n’);/* eat newline */
   if(n<0 || n>value->length) n = 0;  /* no selection */
   value=alcvec(INT,1L);
   *(value->value.Long) = n;
   return(value);
}
```