8 Writing a Recorder Driver

A recorder driver is a SciAn object which controls a video recorder for animation. A custom recorder driver for your own video or film recorder is one of the easiest additions you can write for SciAn.

This chapter describes the process of writing a recorder driver.

Setting up the files

The first thing to do is set up source files and routines to hold your new code. The names of these files and their contents should depend on the model of the recorder driver. Let's say that you are writing a recorder driver for video disc recorder model VDR2000, manufactured by Fonebone industries. You will need to create two files, `ScianVDR2000.c` to contain the code, and `ScianVDR2000.h` to contain external references.

The `ScianVDR2000.c` file needs to contain only two externally callable functions: `InitVDR2000` and `KillVDR2000`. The `Init` function is responsible for creating and registering the recorder driver object. In most cases, the `Kill` function does absolutely nothing. The storage for the recorder driver object is automatically taken care of when SciAn exits, and all the disconnection from the recorder should be done in messages. The `Kill` function is provided just in case some recorder driver should need to clean up some non-object storage.

The `ScianVDR2000.c` file also needs to include all the headers necessary for getting the job done. Unfortunately, there's no easy way of determining in the general case which header files will be needed. Fortunately, a recorder driver does not need much, and the list in this example `ScianVDR2000.c` file should be sufficient:

```
/*Sample recorder driver for the VDR2000.c
*/

#include "Scian.h"
#include "ScianTypes.h"
#include "ScianWindows.h"
#include "ScianObjWindows.h"
#include "ScianDialogs.h"
#include "ScianIDs.h"
#include "ScianErrors.h"
#include "ScianColors.h"
#include "ScianRecorders.h"
#include "ScianVDR2000.h"

#ifdef PROTO
void InitVDR2000(void)
#else
void InitVDR2000()
#endif
```

```
/*Initialize the VDR2000 module, creating and registering the recorder driver*/
{
}

#ifdef PROTO
void KillVDR2000(void)
#else
void KillVDR2000()
#endif
/*Kill the VDR2000 module.  Doesn't do anything.*/
{
}
```

The `InitVDR2000` and `KillVDR2000` functions have been defined. They don't do anything, but that's OK for now. Notice that there are two declarations for each function, one when `PROTO` is defined and one when it is not. Although we haven't yet come across a compiler that didn't recognize prototypes, it's better to be safe than sorry, just in case one day we do. All functions should be declared twice, once prototype-style and once old-style.

Next, the `ScianVDR2000.h` file needs to be created. All this needs is external prototypes for the `Init` and `Kill` functions. Here is an example:

```
/*ScianVDR2000.h
  External stuff for ScianVDR2000.c
*/

#ifdef PROTO
void InitVDR2000(void);
void KillVDR2000(void);
#else
void InitVDR2000();
void KillVDR2000();
#endif
```

Notice once again that the functions are declared twice, once as prototypes, and once as old-style declarations.

Next, you need to change `ScianRecorders.c` so that it calls your `Init` and `Kill` functions. Edit the file and insert an

```
#include "ScianVDR2000.h"
```

at the end of the list of `#include` files. Next, go down to the InitRecorders function, near the end. You will see a group of `Init` calls, which may look something like this:

```
    InitLVR5000();
    InitTQ2026F();
    InitScrDump();
```

Add a call to initialize your recorder driver to this group of calls. If you want to make it the default recorder driver, put it first in the list. The list will then look something like this:

```
    InitVDR2000();
    InitLVR5000();
    InitTQ2026F();
    InitScrDump();
```

Finally, the SciAn `Makefile` needs to be changed to know about your new files. See the "Source files" section in Chapter 6 to find out how to add a source file to SciAn. Once you have done this, make SciAn again. Even though the new source code does nothing, it is satisfying to see that everything goes smoothly.

Note *Some day, we may automate the process of adding new files and routines to get rid of the need to edit existing SciAn modules. Please follow the naming conventions for source files and functions, as this will make it a lot easier for us to automate.*

Implementing the recorder driver

A recorder driver is a single object. The object belongs to a class, from which it inherits variables, controls, and other behavior. It also contains five methods, which perform all the functionality of the recorder driver.

The `Init` routine must create the recorder driver, give it methods and optional variables, and register it with the recorder system.

Creating the object

To create the object, use the `NewRecorder(`*class*`,` *name*`,` *brandName*`)` function. There are currently two possible choices for *class*, both declared in `ScianRecorders.h`: `recorderClass` and `commRecorderClass`. All recorder drivers belong to `recorderClass`, and from this class they inherit a frame rate in frames per second, which is handled by SciAn, and a frame width and height, which can be handled by the recorder driver but is usually ignored. Recorders that also do serial communication must belong to `commRecorder` Class, from which they inherit parameters for communication, such as baud rate.

For our example, we will assume that the VDR2000 is controlled with serial communication and needs therefore to be of class `commRecorderClass`. The first part of the `Init` routine will look something like this:

```
#ifdef PROTO
void InitVDR2000(void)
#else
void InitVDR2000()
#endif
/*Initialize the VDR2000 module, creating and registering the recorder driver*/
{
    ObjPtr recorder;

    recorder = NewRecorder(commRecorderClass, "VDR2000", "Fonebone Ind.");
```

Implementing the methods

Five methods must be given to the new recorder driver: `CONNECT`, `DISCONNECT`, `PREPARETORECORD`, `STOPRECORDING`, and `SNAPONEFRAME`.

The `CONNECT` method connects to the recorder. The message takes no arguments, so the method only takes one argument, the recorder object, and no arguments beyond. It is responsible for establising communications with the recorder, if necessary. It should return `ObjTrue` if successful and `ObjFalse` if not, such as if the recorder is not connected to the machine.

The `DISCONNECT` method disconnects the recorder. The message takes no arguments beyond the recorder object. It is responsible for disconnecting communications and closing I/O files. It should return `ObjTrue` if successful and `ObjFalse` if not, This message will only be sent if the `CONNECT` method succeeded.

The `PREPARETORECORD` method prepares the recorder to record a sequence with a certain number of frames. It takes one argument beyond the recorder object, a long integer giving the number of frames to be recorded. It is responsible for ensuring that there are enough frames left on the recorder and getting the recorder to a state where it is ready to start snapping frames. It should return `ObjTrue` if successful and `ObjFalse` if not, such as if there is not enough space on the recorder for the required number of frames. This message will only be sent if the `CONNECT` method succeeded.

The `STOPRECORDING` method stops recording the current sequence. It takes no arguments beyond the recorder object. It should return `ObjTrue` if successful and `ObjFalse` if not. This message will only be sent if the `PREPARETORECORD` method succeeded.

The `SNAPONEFRAME` method snaps a single frame onto the video recorder. Frames are snapped in sequence from beginning to end. It takes no arguments beyond the recorder object. It should return `ObjTrue` if successful and `ObjFalse` if not. This message will only be sent if the `PREPARETORECORD` method succeeded.

The next thing to do after the recorder driver has been created is to assign the methods. For our example, this part will look something like this:

```
recorder = NewRecorder(commRecorderClass, "VDR2000", "Fonebone Ind.");

SetMethod(recorder, CONNECT, ConnectVDR2000);
SetMethod(recorder, DISCONNECT, DisconnectVDR2000);
SetMethod(recorder, PREPARETORECORD, PrepareToRecordVDR2000);
SetMethod(recorder, STOPRECORDING, StopRecordingVDR2000);
SetMethod(recorder, SNAPONEFRAME, SnapOneFrameVDR2000);
```

The next thing to do is to write the five functions that implement the methods. Remember that these are method definitions, and so must use old-style C declarations, not prototypes. This is so that they can take a fixed number of arguments which is not known at compile time. Also, all methods take at least one argument: the object to which the message is being sent. Other arguments follow.

The `CONNECT` and `DISCONNECT` methods connect to and disconnect from the video recorder, respectively. On a recorder which does not do any serial communication, they usually don't do anything. On a recorder which does do serial communication, they should open and close a communications channel to the device. There are two variables that help this process and are inherited from the control panel. `PORTDEV` is a string variable which contains the name of the device to use for the communications port. `BAUDRATE` is an integer variable, which gives the baud rate of the connection. It has one of the values RB_300, RB_1200, RB_2400, RB_4800, RB_9600, or RB_19200, all of which are declared in ScianRecorders.h.

One thing to remember when writing routines for serial communication is that all communication routines *must* be protected within an #ifdef TERMIO block. If TERMIO is not defined, you are not guaranteed of anything having to do with serial communication.

The following is an example of a basic CONNECT method:

```
static ObjPtr ConnectVDR2000(recorder)
ObjPtr recorder;
/*Connects to the VDR2000.  Returns true iff successful*/
{
#ifdef TERMIO
    int portDev;
    int k;
    ObjPtr var;
    char *devName;
    int baudRate;

    MakeVar(recorder, PORTDEV);
    var = GetStringVar("ConnectVDR2000", recorder, PORTDEV);
    if (!var)
    {
        /*No portdev, what's the point?*/
        return ObjFalse;
    }
    devName = GetString(var);

    MakeVar(recorder, BAUDRATE);
    var = GetIntVar("ConnectVDR2000", recorder, BAUDRATE);
    if (!var)
    {
        /*No baud rate, what's the point?*/
        return ObjFalse;
    }
    baudRate = GetInt(var);

    /*Open a port*/
    portDev = OpenPort(devName, baudRate);
    if (portDev < 0)
    {
        return ObjFalse;
    }

    /*Now that a port is open, do something to talk to it,
     and see if it's there.  This probably involves calling
     the write and read functions with portDev.
     If the recorder is not there, return ObjFalse.*/

    /*  insert code to implement previous comment here   */

    /*If we've fallen through here, the port is OK.  We have to store the
      port for later, so let's use a variable called PORTNUMBER, defined
      in ScianIDs.h*/

    SetVar(recorder, PORTNUMBER, NewInt(portDev));

    return ObjTrue;
#else
    return ObjFalse;
#endif
}
```

Notice that the method is declared as `static`. No other function has any business accessing this routine except through the message mechanism, so it should be local to this file. Also, as all other methods, it is defined using an old-style function header.

The first portion gets the `PORTDEV` and `BAUDRATE` variables. Because the protected GetVar routines `GetIntVar` and `GetStringVar` are used, the return value is guaranteed to be of the correct type or `NULLOBJ`. The function then opens a port using `OpenPort`, and should test to see if the recorder is really there using code not yet written. If it is, the method sets the `PORTNUMBER` variable to the port number and returns ObjTrue.

The `OpenPort` function is not provided by SciAn and must be implemented in the recorder driver. This is where all the magic to set up the serial port goes. The following example sets up a nonblocking serial file with eight data bits and no parity:

```
#ifdef PROTO
static int OpenPort(char *devName, int baudRate)
#else
static int OpenPort(devName, baudRate)
char *devName;
int baudRate;
#endif
/*Opens a port to the recorder, returns it if successful, or <0 if not*/
{
#ifdef TERMIO
    int portDev;
    struct termio ioStuff;
    ObjPtr var;

    portDev = open(devName, O_RDWR | O_NDELAY | O_EXCL);
    if (portDev < 0)
    {
        return portDev;
    }
    ioctl(portDev, TCGETA, &ioStuff);
    ioStuff . c_oflag = 0;
    ioStuff . c_iflag = IGNBRK;
    ioStuff . c_lflag = 0;

    /*Make cflag from baud rate*/
    switch(baudRate)
    {
        case RB_300:
            ioStuff . c_cflag = (unsigned short)
                (B300 | CS8 | CLOCAL | CREAD | CSTOPB);
            break;
        case RB_1200:
            ioStuff . c_cflag = (unsigned short)
                (B1200 | CS8 | CLOCAL | CREAD);
            break;
        case RB_2400:
            ioStuff . c_cflag = (unsigned short)
                (B2400 | CS8 | CLOCAL | CREAD);
            break;
        case RB_4800:
            ioStuff . c_cflag = (unsigned short)
                (B4800 | CS8 | CLOCAL | CREAD);
            break;
        case RB_9600:
            ioStuff . c_cflag = (unsigned short)
                (B9600 | CS8 | CLOCAL | CREAD);
            break;
        case RB_19200:
```

```
            ioStuff . c_cflag = (unsigned short)
                (B19200 | CS8 | CLOCAL | CREAD);
            break;
    }

    ioStuff . c_cc[4] = 1;
    ioStuff . c_cc[0] = 1;

    ioctl(portDev, TCSETA, &ioStuff);
    return portDev;
#else
    return -1;
#endif
}
```

The DISCONNECT method must disconnect from the video recorder, which may include sending it some commands to get it to stop listening. Then it must disconnect the serial communications channel. The following example corresponds to the CONNECT method given previously.

```
static ObjPtr DisconnectVDR2000(recorder)
ObjPtr recorder;
{
    int portDev;
    ObjPtr port;

    port = GetVar(recorder, PORTNUMBER);
    if (port)
    {
        portDev = GetInt(port);

        /*Talk to the recorder to get it to stop listening*/

        /* insert code to implement previous comment here */

        /*Now close the port*/
        ClosePort(portDev);
        SetVar(recorder, PORTNUMBER, NULLOBJ);
    }
    return ObjTrue;
}
```

Notice that this method always returns ObjTrue. If there’s an easy way of telling whether the operation succeeded or not, then it may be appropriate sometimes to return ObjFalse. However, the return values of the DISCONNECT and STOPRECORDING methods are not really important.

Fortunately, the helper routine ClosePort is a lot simpler than OpenPort:

```
#ifdef PROTO
static void ClosePort(int port)
#else
static void ClosePort(port)
int port;
#endif
/*Closes port*/
{
#ifdef TERMIO
    close(port);
#endif
}
```

Until this point, exactly how to do serial I/O has been somewhat glossed over. From here on, it will become more important. The example CONNECT method resulted in a file, whose integer UNIX ID was stored in the variable PORTNUMBER. The basic method of doing I/O is to get this port number and use it as a UNIX file in a read or write call. The example code set up the port to be nonblocking, so a read will return immediately without waiting for a character.

Beyond that, there are a large number of strategies that can be used for serial communications, including buffers, timeouts, etc. You are on your own in designing your own strategy. Make sure that you do not block forever.

The approach used by existing SciAn recorder drivers is simple-minded, but effective. Characters are sent individually and the process blocks while waiting for a response. However, there is always a timeout, which will return control to SciAn after a while, no matter what. If you want to adopt this approach, look in ScianLVR5000.c for ideas. There are, however, many better approaches, and we encourage you to come up with one if you are so inclined.

Once you have decided on your strategy, you can go ahead and implement the PREPARETORECORD, STOPRECORDING, and SNAPONEFRAME methods. The implementation of these is totally dependent on the video recorder, so only skeleton examples are presented here:

```
static ObjPtr PrepareToRecordVDR2000(recorder, nFrames)
ObjPtr recorder;
long nFrames;
/*Prepares to record nFrames on a VDR2000.  Returns true iff successful.*/
{
    ObjPtr port;

    port = GetVar(recorder, PORTNUMBER);
    if (port)
    {
        int portDev;

        /*Get the actual port*/
        portDev = GetInt(port);

        /*Now talk to the recorder and tell it to find an area big enough
          to record nFrames.  If there is an area, set it to get ready to
          record and return ObjTrue.  If there is no such area, return
          ObjFalse.*/

        /* insert code to implement the previous comment here */
    }
    else
    {
        return ObjFalse;
    }
}

static ObjPtr StopRecordingVDR2000(recorder)
ObjPtr recorder;
/*Stops recording on a VDR2000.  Returns true iff successful.*/
{
    ObjPtr port;

    port = GetVar(recorder, PORTNUMBER);
    if (port)
    {
```

```
        int portDev;

        /*Get the actual port*/
        portDev = GetInt(port);

        /*Now talk to the recorder and tell it to stop recording.  Return
          ObjTrue if successful, ObjFalse if not.*/

        /* insert code to implement the previous comment here */
    }
    else
    {
        return ObjFalse;
    }
}

static ObjPtr SnapOneFrameVDR2000(recorder)
ObjPtr recorder;
/*Snaps one frame on a VDR2000.  Returns true iff successful.*/
{
    ObjPtr port;

    port = GetVar(recorder, PORTNUMBER);
    if (port)
    {
        int portDev;
        int width, height;

        /*Get the actual port*/
        portDev = GetInt(port);

        /*Get the width and height of the screen*/
        GetScreenSize(&width, &height);

        /*Now talk to the recorder and tell it to snap the next frame in
          the sequence.  Return ObjTrue if successful, ObjFalse if not*/

        /* insert code to implement the previous comment here */
    }
    else
    {
        return ObjFalse;
    }
}
```

Note that the final method, `SnapOneFrameVDR2000`, uses the `GetScreenSize` function to get the width and height of the recording frame. This function simply gets the appropriate variables from the current recorder driver. This is mostly for recorder drivers that save to image files. Most drivers for recorders that snap directly off the video can ignore these numbers, although a particularly clever one might, for example, automatically zoom in a scan converter to the selected area.

Registering the recorder driver

Once the object has been created and its methods have been set, it must be registered as a recorder driver with a single `RegisterRecorder` call at the end of the Init function, like this:

```
    RegisterRecorder(recorder);
}
```

This function adds the recorder driver to the SciAn database and restores any parameters saved in a file. From then on, everything will work automatically. Messages will be sent to your object when it is time to record.

9 Writing a File Reader

< under construction >

References

Brookhaven National Laboratories, 1992. *Protein Data Bank Atomic Coordinate and Bibliographic Entry Format Description*.

Buning, P., Pierce, L., and Elson, P., 1990. *PLOT3D User's Manual, Version 3.6*. NASA Technical Memorandum 101067.

Haines, E. A. 1987. "A Proposal for Standard Graphics Environments," *IEEE Computer Graphics and Applications*, **7(11)**:3-5.

National Center for Supercomputing Applications, 1990. *NCSA HDF Calling Interfaces and Utilities*, University of Illinois, Urbana-Champaign.

Walatka, P. P., Plessel, T., McCabe, R. K., Clucas, J., and Elson, P., 1991. *FAST User's Manual, Beta 2.0.* NASA Ames Research Center.

Weast, R. ., Lide, D. R., Astle, M. J., and Beyer, W. H., 1989. *CRC Handbook of Chemistry and Physics.* CRC Press, Inc., Boca Raton, F-187.