Video Object — A Library for Controlling Video Devices

Wee Lee Lim
George D. Drapeau
MAEstro Project
Stanford University
maestro@sioux.stanford.edu

October 30, 1992

Contents

1 Introduction

For a single application to provide support for a variety of computer-controlled videodisc and videotape players, the application needs to include serial-line drivers for a potentially large number of those devices. For this purpose, we have created a generic "Video Object" that supports features common to a number of video devices.

The purpose of this file is to instruct application programmers how to use the Video Object library in their own applications, and to provide help in adding new devices to the library.

If you find this object useful and write support for players other than those supported here, we would be grateful if you would send us a copy of your source code so that we can add your driver to the publicly distributed VideoObject.

We hope that you find this code useful in your applications. If you have any comments, questions, suggestions, or source code you would like to add, please send mail to `maestro@sioux.stanford.edu`.

2 Objectives

The Video Library was created with the following objectives :

1. To allow programmers to write applications that support a variety of video devices (such as videodisc and videotape players) without having to write device-specific code. The application program calls a generic `Dev<Function>` regardless of what device is being used, and the video library is able to call the correct `<Function>` for the particular device currently being used.
2. To allow programmers to be able to easily add on to the suite of device drivers currently available.

3 Definition of the Video Object

The Video Library consists of device-specific code to support a number of video devices. A generic "object" is provided to the application programmer; this structure contains pointers to functions that represent the device specific routines for common actions such as 'play', 'pause', 'stop', 'search', etc. In addition to these function pointers, the object contains information about the currently-used video device (such as the serial line used for communication with the device, the serial line attributes for the device, its performance characteristics, and so on). This information (device 'configuration') is stored in the data structure called `Config`, which is stored as part of the Video Object.

This section will describe the following components of the Video Object in detail:

- Config — Data structure containing device-specific serial line and performance attributes
- Video Functions — The various functions supported by the Video Object and their semantics.

This section ends with a note on status codes, outlining the differences between the status codes returned by the devices, and the status codes that are returned from the Video Object to the application.

3.1 The `Config` Structure — Video Device Configuration

The `Config` data structure stores the configuration of a particular video device, and is defined as follows:

```
typedef struct _Config
{
  char modelName[MaxNameLength];
  char serialPort[15];
  int baudRate;
  int eolChar;
  int charSize;
```

```
        enum Parity parity;
        enum Boolean igParError;
        enum Boolean flowControl;
        int maxSpeed;
        int fd;
    } Config;
```

The `modelName` field is a String that denotes the name of the device. This will be the name used by the application and the library.

The `serialPort` field is a String that denotes the serial port to which the device is physically attached. This field is likely to be different for different machines (and users). To let applications to use a different value for this field easily without having to change and recompile their code, the Video Object calls `XGetDefault()` to determine the user's default serial port to use for the device.

`XGetDefault()` is called with the `modelName` field as its second argument, and `serialPort` as its third argument. Therefore the form for storing serial port defaults as an X resource is as follows:

```
<modelName>.serialPort:          <serial port>
```

The `baudRate` field is an Integer that denotes the baud rate at which data transfer is carried out.

The `eolChar` field is an Integer that represents the ASCII value (decimal) of the character sent by the device denoting the end of a line and that a read() should be performed.

There are two reasons why the `eolChar` field is necessary. Firstly, this field distinguishes whether the Video Object should use canonical input processing (the value of `eolChar` is non- negative) or non-canonical input processing (the value of `eolChar` is negative) to communicate with the device. Canonical input processing is used for devices that return a special character to denote the end of a line, while non-canonical input processing is used for those devices that do not return a special end-of-line character. [For more information, please read the man pages on termio(4). For an example of a device that does canonical input processing, please look at the source code for the Pioneer4200 Video Object (Pioneer4200Driver.c). For an example of a device that uses non-canonical input processing, please look at the source code for the Sony1550 Video Object (Sony1550Driver.c).]

Secondly, with canonical input processing, Unix recognizes a line by the `NEWLINE (ASCII LF)` character or the `EOF (ASCII EOT)` character, or two other user-specified characters. Because different devices have different end-of-line characters, the Video Object requires that the device specifies explicitly its end-of-line character in this `eolChar` field.

The `charSize` field is an Integer that denotes the size in bits of a data character. Valid values can be 5, 6, 7 or 8.

The `parity` field is an Enum Parity that can be either `Odd`, `Even`, `NoParity` or `Any`.

The `igParError` field is an Enum Boolean that is `Yes` if the Video Object wants to ignore parity errors, and `No` if the Video Object does not want to ignore parity errors.

The `flowControl` field is an Enum Boolean that is `Yes` if the flow control is desired for the device with which the Video Object is communicating, and `No` if the device does not use flow control.

The `maxSpeed` field is an Integer that denotes the maximum speed in terms of frames/second that the device is capable of playing.

The `fileDescriptor` field is an Integer that denotes the file descriptor for the device, after the serial port connection has been made. Its initial value can be set to any negative number.

3.2 Notes on Configuration Settings

1. If the device does not return an end-of-line character, enter a negative number in the `eolChar` field. It is assumed that non-canonical input processing will be used if a negative number is entered for the end-of-line character. See description of `eolChar` field above.

2. In non-canonical input processing, the initial values for `MIN` and `TIME` are:

```
c_cc[VMIN] = 1 and c_cc[VTIME] = 0
```

This means that the read() function blocks until at least 1 character is received. Please read the man pages on `termio(4)` for more details.

Here are two sample device configurations, both taken from the `videoObjects.c` file for the VideoEdit application. The first describes the Pioneer 4200 videodisc player. The second configuration describes the Sony 1550 videodisc player. A table for ASCII values can be found in A.

```
static Config Pioneer4200Config =
{
  "Pioneer4200",                 <= modelName
  "/dev/ttya",                   <= serial Port
  4800,
  10,                            <= NL (ASCII 10) is the eolChar
  8,                                             for Pioneer4200
  Even,
  Yes,
  120,
  -1
  }
  ;

static Config Sony1550Config =
{
  "Sony1550",
  "/dev/ttyb",
  2400,
  -1,                            <= non-canonical input processing
  8,
  Even,
  Yes,
  90,
  -1
  }
  ;
```

3.3 Notes on Changing the Configuration

`videoObjects.c` is the file that stores the configurations for each device in an application (see 5.1). To use the Video Library, copy this file into your own directory and compile it with the rest of your application. Any changes you wish to make to video device configurations can be made in your copy of the `videoObjects.c` file. If you are using one of the already-supplied devices, the only setting you might likely change is the serial port to which the device is connected, and even this may not be necessary.

See the end of the `videoObj.h` file for a list of device configurations for currently-supported devices.

The serial line is set by `SetSerialLine()`, a function supplied by the Video Library. `SetSerialLine()` has two arguments:

1. Device configuration structure
2. Serial port

Because the `serialPort` field is likely to differ from machine to machine, the Video Object chooses the serial port to connect with by using the following procedure:

- Use the second argument to `SetSerialLine()` if it is not NULL. `SetSerialLine()` should, in general, be called with NULL as the second argument, because in the most cases, the serial port is not explicitly chosen. The second argument should be non-NULL, only when the user specifies from within the application a particular serial port to use.
- Call `XGetDefault()`. If it returns a non-NULL value for the serial port for this device, then use this value.
- Use the value set in the configuration structure.

Thus, to change serial port without modifying any code, it is easiest to set the default serial port in `/.Xdefaults`.

Here is an example X resource showing how a user might set the default serial port used by the Pioneer4200 device driver:

```
Pioneer4200.serialPort:      /dev/ttya       <= Pioneer4200 player
                                                is on /dev/ttya
```

3.4 Definition of Video Object Functions

The Video Object has at present 23 basic functions. Not all devices will be able to perform all 23 functions; such devices will have NULL function pointers for those functions they cannot perform.

The `VideoObject` data structure is defined as follows:

```
typedef struct _VideoObject
{
  Config* DevConfig;
  int (*DevPlayPtr) (VideoObject*);
  int (*DevPlayFromToPtr) (VideoObject*, int, int, int);
  int (*DevFastForwardPtr) (VideoObject*);
  int (*DevReversePtr) (VideoObject*);
  int (*DevCalcSpeedPtr) (VideoObject*, int, int);
  int (*DevPlayAtSpeedDirPtr) (VideoObject*, int, enum Direction);
  int (*DevStepPtr) (VideoObject*, enum Direction);
  int (*DevStillPtr) (VideoObject*);
  int (*DevStopPtr) (VideoObject*);
  int (*DevSetDefaultsPtr) (VideoObject*, int, int, int, int);
  int (*DevSetAudioPtr) (VideoObject*, int);
  int (*DevSetVideoPtr) (VideoObject*, int);
  int (*DevSetAddModePtr) (VideoObject*, int);
  int (*DevSetAddressDisplayPtr) (VideoObject*, int, int);
  int (*DevEjectPtr) (VideoObject*);
  int (*DevPowerPtr) (VideoObject*, int);
  int (*DevQueryFramePtr) (VideoObject*);
  int (*DevQueryChapterPtr) (VideoObject*);
  int (*DevQueryAudioPtr) (VideoObject*);
  int (*DevQueryVideoPtr) (VideoObject*);
  int (*DevQueryMediumPtr) (VideoObject*, char*);
  int (*DevQueryStatusPtr) (VideoObject*);
  int (*DevPingPtr) (VideoObject*);
  int (*DevRecordPtr) (VideoObject*);
  int (*DevRecordFromToPtr) (VideoObject*, int, int, int);
} VideoObject;
```

By compounding different functions, applications will be able to achieve most of the operations commonly desired of a video device. The following are the descriptions of what each function should do.

The functions return a status code, unless otherwise stated. For a description of the status codes, please see 3.5.

1. `int (*DevPlayPtr) (VideoObject* theObject)`

   Pointer to function that sets `theObject` in 'playback' mode.

2. `int (*DevPlayFromTo) (VideoObject* theObject, int startAddress, int endAddress, int speedInFramesPerSecond)`

   Pointer to function that plays a segment, from `startAddress` to `endAddress`.

   `startAddress` and `endAddress` are in terms of frames if the player is in frame mode, and in terms of chapters if the player is in chapter mode.

   For devices that have indices rather than chapters, and only have relative search of indices, `endAddress` is used to indicate direction. As usual, 0 is Forward, 1 is Reverse.

   This function is used in 4 different ways:

   (a) `startAddress == endAddress`
   Searches to `startAddress` and stills. This is a request for a non-blocking search; in other words, `DevPlayFromTo` will send the search command to the specific device then will return as soon as possible (for some players, it is possible to read an acknowledge message indicating that the command has begun execution, then a completion message indicating that the command has completed execution. The VISCA protocol follows this type of communications).

   (b) `startAddress != NULL, endAddress == NULL`
   Searches to `startAddress` and stills. This is a request for a blocking search; in other words, the `DevPlayFromTo` function will wait until the device completes its search before returning to the calling application.

   (c) `startAddress == NULL, endAddress != NULL`
   Play from current position (no search) until `endAddress` at `speedInFramesPerSecond`.

   (d) `startAddress != endAddress, startAddress != NULL, endAddress != NULL`
   Play from `startAddress` to `endAddress` at `speedInFramesPerSecond`. However, if `startAddress` is greater than `endAddress`, then the function should return an error.

3. `int (*DevFastForwardPtr) (VideoObject* theObject)` Pointer to function that plays forward at higher than normal (device-dependent) speed.

   It may continue to play at this speed until some other command is issued (the usual case for VCRs), or it may return to normal playback after scanning a certain number of frames (usually the case for videodisc players).

4. `int (*DevReversePtr) (VideoObject* theObject)` Pointer to function that plays reverse at higher than normal (device-dependent) speed.

   It may continue to play at this speed until some other command is issued (the usual case for VCRs), or it may return to normal playback after scanning a certain number of frames (usually the case for videodisc players).

5. `int (*DevCalcSpeedPtr) (VideoObject* theObject, int inputValue, int playMode)` Pointer to function that returns speed (in number of frames per second) that can be played by `theObject`.

   `inputValue` is the actual speed in frames per second at which the user would like to play. `playMode` is 0 for normal playback and 1 for segment play (ie playback from one address to another), since it is found in some cases that devices are able to play at special speeds only in normal playback mode.

6. `int (*DevPlayAtSpeedDirPtr) (VideoObject* theObject,`
   `int speedInFramesPerSecond,`
   `enum Direction direction)`

   Pointer to function that plays at `speedInFramesPerSecond` in `direction`, where `direction` can be `Forward` or `Reverse`.

   This function should call `DevCalcSpeed()` so as to obtain the device-specific speed based on `speedInFramesPerSecond`.

7. `int (*DevStepPtr) (VideoObject* theObject, enum Direction direction)`

   Pointer to function that steps one frame in `direction`.

8. `int (*DevStillPtr) (VideoObject* theObject)`

   Pointer to function that puts `theObject` into `still` mode, meaning the frame is frozen, but video is not muted.

   This function should be able to execute in any mode, which means that if segment play (ie playback from one address to another) is not interruptible by a normal pause command, an interrupt command should be used to implement pause.

   An example of a device that cannot pause segment play is the NEC PC-VCR. In order to implement a pause in this mode, the driver sends out an interrupt command before it sends out its pause command.

9. `int (*DevStopPtr) (VideoObject* theObject)`

   Pointer to function that puts `theObject` into `stop` mode, meaning that playback is stopped and video is muted.

   This function does NOT shut down the disc motor. This function should be able to execute in any mode, which means that if segment play is not interruptible by a normal stop command, an interrupt command should be used to implement stop. See similar comments for `DevStillPtr`, above.

10. `int (*DevSetDefaultsPtr) (VideoObject* theObject, int audio,`
    `int addressingMode,`
    `int addressDisplayOnOff, int displayMode)`

    Pointer to function that puts `theObject` into desired mode when starting it up.

    - `audio`
      One of `Mute, Left, Right, Stereo`. Others can be assigned by the programmer for other modes, but this might not be implemented uniformly across all video objects!
    - `addressingMode`
      Either `PlayerFrameMode = Normal` (usually means frame addressing) or `PlayerChapterMode = Indexing` (or Chapters).
    - `addressDisplayOnOff`
      Either `FeatureOff` (No address display on screen) or `FeatureOn` (Address display on screen).
    - `displayMode`
      One of `DisplayFrame` (Display frame address on screen), `DisplayChapter` (Display chapter on screen), or `DisplayFrameAndChapter` (Display frame and chapter on screen).

    The `#define's` for these parameters are in `PlayerStatus.h`.

11. `int (*DevSetAudioPtr) (VideoObject* theObject, int mode)`

    Pointer to function that sets audio to `mode` (see definition of `DevSetDefaultsPtr`, above).

12. `int (*DevSetVideoPtr) (VideoObject* theObject, int mode)`

    Pointer to function that sets video to `mode`, either `FeatureOn` or `FeatureOff`.

13. `int (*DevSetAddModePtr) (VideoObject* theObject, int mode)`

    Pointer to function that sets addressing mode to `mode` (see definition of `DevSetDefaultsPtr`, above).

14. `int (*DevSetAddressDisplayPtr) (VideoObject* theObject, int onOff, int mode)`

    Pointer to function that either turns on-screen address display to `on` or `off` depending on value of `onOff` and sets the display to `mode` (see definition of `DevSetDefaultsPtr`, above).

15. `int (*DevEjectPtr) (VideoObject* theObject)`

    Pointer to function that ejects the medium from the player. Returns status code.

16. `int (*DevPowerPtr) (VideoObject* theObject, int mode)`

    Pointer to function that turns power either on or off, depending on `mode`.

17. `int (*DevQueryFrame) (VideoObject* theObject)`

    Pointer to function that asks the player for the current address of the medium. Returns the address in terms of frames.

18. `int (*DevQueryChapter) (VideoObject* theObject)`

    Pointer to function that asks the player for the current chapter of the medium. Returns the chapter number.

19. `int (*DevQueryAudio) (VideoObject* theObject)`

    Pointer to function that asks the player for status of audio output. Returns audio status.

20. `int (*DevQueryVideo) (VideoObject* theObject)`

    Pointer to function that asks the player for status of video output. Returns video status.

21. `int (*DevQueryMedium) (VideoObject* theObject, char* result)`

    Pointer to function that asks the player for medium ID, where the medium is either a videotape or a videodisc. Returns the medium ID if integer, or if a string, returns medium ID in `result`, and returns 0.

22. `int (*DevQueryStatus) (VideoObject* theObject)`

    Pointer to function that asks the player for its status, eg play, pause, search, etc.

23. `int (*DevPing) (VideoObject* theObject)`

    Pointer to function that sends a command to the player that is guaranteed to elicit some response from the player. It returns the number of characters read. This function is used only during the installation of a driver, by the Video Library function `PlayerVerified()` so that the application can attempt to verify that if the player is correctly selected.

    The verification is not guaranteed to be foolproof, because a command may be recognized by many different players. The programmer can make the Ping() function more intelligent by returning a positive number of characters read only when the response corresponds to what the current player will return. This eliminates some cases of misidentification, but is again not foolproof and it can be difficult to enumerate all possible responses in the case of some drivers.

24. `int (*DevRecord) (VideoObject* theObject)`

    Pointer to function that begins normal recording on the player. In video terminology, this is known as "assemble edit".

25. `int (*DevRecordFromTo) (VideoObject* theObject, int startAddress,`
    `int endAddresss,`
    `int speedInFramesPerSecond)`

    Pointer to function that records a segment of video, from `startAddress` to `endAddress`. This function is used for insert editing, and is currently used in 2 different ways:

    (a) startAddress == NULL, endAddress == NULL
        Puts player into "Insert Edit" mode, allowing precise time-code recording to occur.

    (b) startAddress == 0, endAddress != 0
        Records from current position (no search) until `endAddress` is reached.

    Notes on DevRecordFromTo:

    (a) Although `speedInFramesPerSecond` field is here, as with `DevPlayFromTo()`, it is unlikely that players will allow recording at other than 1x speed. This field is here to accommodate the possibility of future devices that can record at other than normal speed.

    (b) To do insert editing from, for example, frame 100 to frame 500, use the VideoObject by first calling `DevPlayFromTo(theObject, 100, 100, 0)`. Next, call `DevRecordFromTo(theObject, 0, 0, 0)` to turn on the insert editing function of the player. Finally, call `DevRecordFromTo(theObject, 0, 500, 0)` to do the actual recording.

        It was decided to separate the semantics of putting a deck into insert edit mode from the semantics of actually doing the recording in order to help achieve better synchronization between playback and record decks. Some devices suffer a long setup time when requested to be put into insert edit mode; by separating the two functions, application programmers can better schedule editing functions for synchronized use.

    (c) To record a single frame of video (e.g., frame 100), call `DevRecordFromTo()` in this manner: `DevRecordFromTo(theObject, 100, 100, 30)`.

3.5 Status Codes and Error Codes

Almost all the functions above return a status code. These codes are listed in `PlayerStatus.h` so that the application program gets a uniform status code for a particular condition across all the devices.

If you are writing code to support a new video device, please make sure that you map whatever status codes your device returns into the common status codes defined in `PlayerStatus.h`. That way, the application programmer will only need to code for one set of status codes.

Sometimes, the device will return an error code. If you are writing code to support a new video device, you should write the function within the driver that writes commands out to the serial line and waits for acknowledgments from the serial line such that it always detects when an error is returned. It then should decode the error and either display the appropriate error message, by calling `DisplayError()` or take the appropriate corrective measures. The error code should not be returned to the application. Instead, use `PlayerReturnError`.

4 Writing an Application to use the Video Library

This section explains how your application can use the Video Library.

4.1 Using the Video Object Functions

The application uses the generic Video Object functions to perform the desired function for a particular object, by passing into these generic functions a pointer to the instance of the particular video object. The application must therefore obtain this pointer somehow.

The way the application does this is by calling `DevInit()`, passing in the name of the device to be controlled. `DevInit()` will return a pointer to an instance of the device.

It may be necessary for your application to control several devices at a time; to do so you may repeatedly call the `DevInit()` function with the appropriate parameters as necessary to create as many video device controllers as your application needs. If any two use the same serial port, then these two control the same physical device. If they are set up with different serial ports (this can be done after `DevInit()` has set up the object, by calling `SetSerialLine()`), then they control different devices.

The actual function called depends on the video object passed to the generic `Dev<Function>` calls.

4.2 Status codes

In order that applications get a uniform status code for a particular condition, some common status codes have been defined in `PlayerStatus.h` that will allow the application to check the status of the player. However, not all devices will be able to return all status codes.

The application program does not need to check for any error code from the driver in any situation, since errors will be reported to the application and displayed using `DisplayError()`. However, the application may check for the status of the device and anticipate if an error will occur (by knowing what is permissible and what is not, for example, the NEC PC-VCR cannot pause unless in playback mode), then take steps to help avoid the error (using the above case as an example, the application can set the NEC PC-VCR into playback mode before calling the `DevPause()` function).

4.3 Additions to the Application's Makefile

1. Include `videoObjects.c` in `SOURCES` or `SOURCES.c`.
2. Define `VideoObjectSourceDir` as the directory where the Video Library files are stored.
3. Add `$(VideoObjectSourceDir)/VideoLib.a` to `LDLIBS` before the X library (`-lX` or `-lX11`).
4. Add `-I$(VideoObjectSourceDir)` to `CPPFLAGS`.

   Note: Remember to declare the 4 functions in `videoObjects.c` in your application.

4.4 Changes to `videoObjects.c`

1. Include the header files of your application.
2. Modify `DisplayError()` and `DisplayChoice()` for the application.

Recompile!

5 Adding New Drivers to Your Application

This section is for programmers who wish to write their own serial line drivers. It explains the various files in the video library, and sets out the steps to take when writing a new video object for the library for use in an existing application.

5.1 Files in the Video Library

The Video Library is made up of the following files. **None** of them should be changed, unless you **really** know what you are doing.

- `videoObj.c`

  This file contains the core functions to :

1. Set up the video object for use by an application program.
   `VideoObject* BuildVideoObject(VideoObject* deviceType)` returns an instance of a video object for a device of `deviceType`.
2. Set the serial line for the device. `int SetSerialLine(Config* configuration, char* newSerialPort)` returns a file descriptor for the device set to `configuration`.
3. Check if a particular function has been implemented for a particular object.
   `enum Boolean CheckImplemented(VideoObject* anObject, int aFunction)` returns either `Yes` or `No`, depending on whether `anObject` has `aFunction` implemented or not. `aFunction` is `#defined` in `videoObj.h`.

   This function is useful when an application program must decide when to provide a function that is not common for a specific device.
4. Test if the player and serial port are correctly selected. `int PlayerVerified(VideoObject* theObject, char* serialPort)` returns either `1` (player and port are OK), or `-1` (either the player or the serial port is wrong).

   As this function uses the `DevPing` function to verify the device and `DevPing` is not foolproof (see description of DevPing in 3.4), this test is not guaranteed to be correct in all cases.
5. Direct the generic `Dev<Function>` calls made to the correct `<Function>` for the particular device:
   `int Dev<Function>(VideoObject* theObject, <OtherParameters>)`
6. Perform default `<Function>` for non-implemented `Dev<Function>` calls.
   `int Default<Function>(VideoObject* theObject, <OtherParameters>)`
7. Get the maximum speed in frames per second that a device can play:

   `int GetDeviceMaxSpeed(VideoObject* theObject)`

- `videoObj.h`

  Header file for the Video Object, containing the relevant files, declarations, data structures and data types.

- `PlayerStatus.h`

  Header file for definitions of status codes returned by the device.

- Device Driver.c files

  These are files containing the code for controlling the devices currently supported.

- Device Driver.h files

  These are the header files for the corresponding Driver files, containing the declarations for the Video Object functions, and the definitions for the error codes recognized by the particular driver. Also, specific status codes for the device that is used **only internally** within the driver code can be specified here, in addition to the status codes provided in `PlayerStatus.h`. These codes should not clash with the codes defined in `PlayerStatus.h`!

- Makefile

  A Makefile is supplied with the video library that compiles the component drivers and builds a library out of it.

5.2 Stub Files and Functions for Use by the Video Library

The Video Library supplies a stub file that contains 4 stub functions to aid the use of the Video Objects by the application programs, as well as the configuration of each Video Object, and the structures of the Video Objects.

The stub file is `videoObjects.c`, and the 4 stub functions are:

1. `VideoObject* DevInit(char* deviceName, char* serialPort)`
   Matches `deviceName` with the correct static variable defining the appropriate videoObject, and returns a pointer to an instance of the desired video object, by calling `BuildVideoObject()`. Then it calls `SetSerialLine()` to set the serial port to `serialPort`.

2. `void DisplayError(char* errorMsg1, char* errorMsg2)`
   Takes the 2 strings returned by the driver that explains the error, and displays it appropriately to the user. Application programs should rewrite this function to best suit their needs. Two strings are provided in the case of long error messages.

3. `void PrintDiagnostics(char* msg)`
   If global enum Boolean variable `diagMode` is `Yes`, then the diagnostic messages from the application and the drivers will be printed (either to standard output, or wherever you might want to send the messages). This is a useful feature, because the applications can hang easily if the wrong messages were sent, or if there are timing problems, and the diagnostics from the drivers would help identify the cause of any device-related problems.

4. `int DisplayChoice(char* msg1, char* msg2,`
   `char* choiceMsg1, char* choiceMsg2)`
   Takes the 2 strings returned by the driver that explains the choice to be made, and displays it appropriately to the user. The choice made is returned as an integer. Application programs should rewrite this function to best suit their needs. Two strings are provided in the case of long explanation messages.

5.3 Incorporating the New Video Object into the Video Library

This section describes how to add a new video device to the Video Library.

When you have written your own driver code and wish to add it to the Video Library, do the following to add the driver to the library:

1. Add `<DeviceName>Driver.c` and `<DeviceName>Driver.h` to `SOURCES` in the Video Library's Makefile.

2. Add `<DeviceName>Driver.o` to `OBJECTS` in the Video Library's Makefile.

3. Recompile the Video Library.

5.4 Incorporating the New Video Object into the Application

This section describes how to add the new device supported into your own application. The changes to be made are all in `videoObjects.c`.

1. Include the appropriate header file for your device. For example, for the Pioneer4200 player:

```
    #include <Pioneer4200Driver.h>
```

2. Add the configuration of your new object as a static variable, named `<modelName>Config`. If you want to keep a copy of the configuration as reference, copy the configuration as a **comment** at the end of `videoObj.h`.

   An example with the Pioneer4200 player:

```
static Config Pioneer4200Config =
{
  "Pioneer4200",
  "/dev/ttya",
  4800,
  13,
  8,
  Even,
  Yes,
  120,
  -1
  }
  ;
```

3. Add the name of your device (as it is in the `modelName` field) into the `allDevices` array and increment `numDevices` by 1.

4. Write the video object for the device, naming the static variable `<modelName>Obj`. For the functions that are not implemented for the device, put a `NULL` in its place.

5. In function `DevInit`, add another else-if statement to call `BuildVideoObject()` with `&<modelName>Obj` if `deviceName` matches `<modelName>`.

Naming convention: Name your driver code `<DeviceName>Driver.c` and its header file, `<DeviceName>Driver.h`. The device name should be unique.

5.5 Notes on Writing the Video Object

- Implement each relevant video object function for the new video device, following the description of what each function is supposed to do in 3.4 above.
- The functions in `<DeviceName>Driver.c` that read and write from the serial line should take care of error handling, and if error display is necessary, should call `DisplayError()` in `videoObjects.c` while still in `<DeviceName>Driver.c`. Do not return error codes specific to each device back to the application program. These should only be used within `<DeviceName>Driver.c` and defined in `<DeviceName>Driver.h`.
- To aid tracking down of communication problems with the device, the functions that send and receive commands from the serial line should call `PrintDiagnostics()` (implemented in `videoObject.c`) with descriptive messages about what is being sent and received, so that if so desired, the user can see the communication between the device and the computer.
- Please use the status codes in `PlayerStatus.h` when returning from the Video Object functions. Do not use any other status codes that you define yourself. If a status code is returned that is not in `PlayerStatus.h`, please pick the closest match. Please see 3.5 above.
- In `<DeviceName>Driver.h`, include `PlayerStatus.h` and `videoObj.h`.

Recompile!

6 Enumerated Types

The following enumerated types are defined in `videoObj.h` and can be freely used in any application using the Video Library.

```
        enum Parity {Odd, Even, Any};
        enum Boolean {No, Yes};
        enum Direction {Forward, Reverse};
```

A ASCII value table

ASCII value table (Decimal) - Character

0 NUL	1 SOH	2 STX	3 ETX	4 EOT	5 ENQ	6 ACK	7 BEL
8 BS	9 HT	10 NL	11 VT	12 NP	13 CR	14 SO	15 SI
16 DLE	17 DC1	18 DC2	19 DC3	20 DC4	21 NAK	22 SYN	23 ETB
24 CAN	25 EM	26 SUB	27 ESC	28 FS	29 GS	30 RS	31 US
32 SP	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 DEL