OSPREY

Video for Windows NT/2000/XP SDK

Version 2.1.0

P/N 40-02055-01
ViewCast.com
Osprey Technologies Division
11/7/2001

TABLE OF CONTENTS

INTRODUCTION 1
Improvements of SDK Version 2.1.0 Over Previous Versions 3
Support and Feedback 5
Background: the Bt848/878 6
To Use This Kit You Need… 7
Basic Techniques 8

COMPONENTS OF THE SDK 11
Accessing Multiple boards 12
Option 1: 13
Option 2: 13
Option 3: 13
App2Ctl: 14
Background: msvideo Registry Keys – Windows NT 4.0: 14
Background: msvideo Registry Keys – Windows 2000/XP: 15
Specific Osprey Driver Behavior – Hunt Mode: 16
Specific Osprey Driver Behavior – Setting up Registry Keys: 17
Controlling the Video Source - CtlApp 18
Controlling the Video Source from a Separate Process - XCtlApp 20
Accessing Closed Caption Data – “Generation-2” Method - CC2Streams 22
Accessing Closed Caption Data – “Generation-2” Method - CC2Screen 24
Accessing Closed Caption Data – “Generation-2” Method - CC2Ctl 26
Sample App 26
Accessing Closed Caption Data – Old Method - CcApp 27
Sample App 28
VBI Raw Capture - VbiGraph 29
Logo Control - LogoApp 31
Cropping Control – CropApp 33
Controlling Audio Settings - vfwMixer 35
Accessing the Bt848/878 registers - RegApp 36
RegAdmin 37
Controlling Advanced Video Processing 44
Analog Bypass (Osprey-500 and Osprey–2000) 44
Filtering 44
Odd/Even Send 44
Force Odd/Even Field First 45
Sample Application: advApp 45

FUNCTIONS 47
OpenDriver() 48
Sample usage: 48
Parameters: 48
Return Values: 49
CloseDriver() 50
Sample Usage: 50
Parameters: 50
Return Value: 50
SendDriverMessage() 51
Sample Usage: 51
Parameters: 51

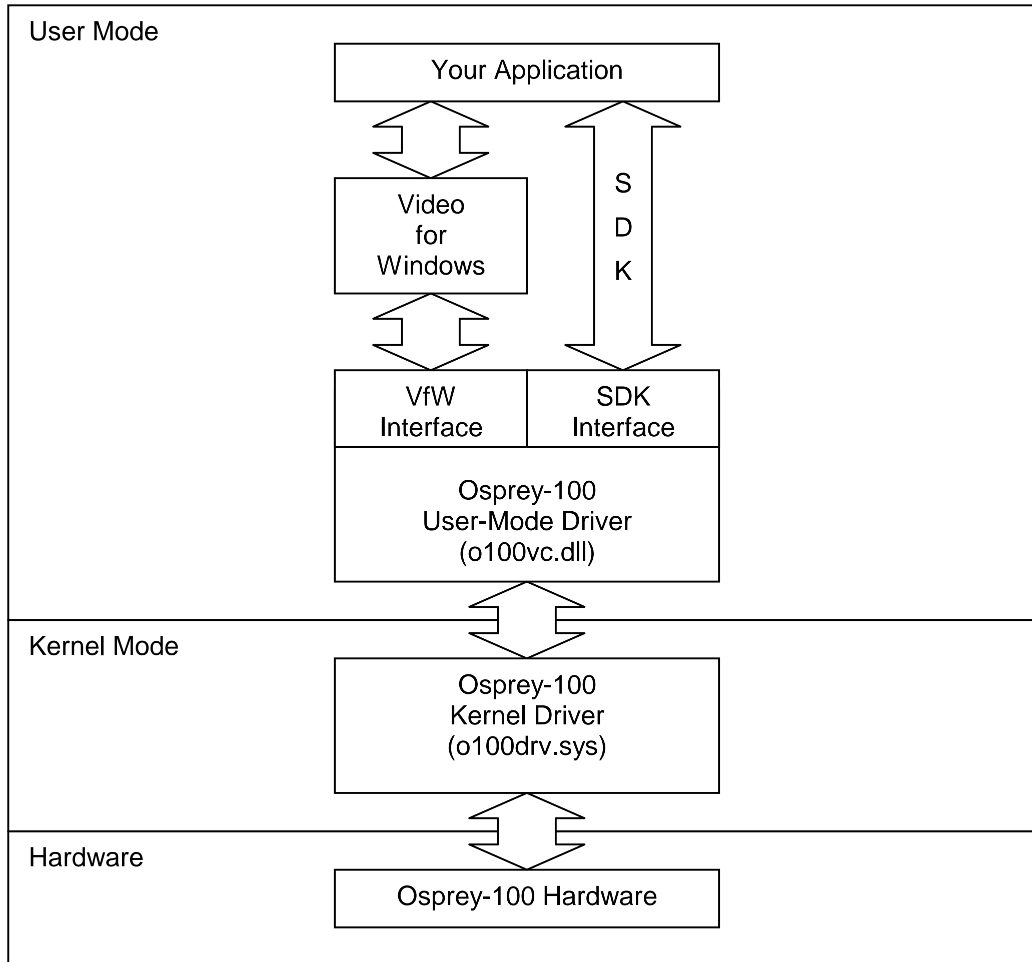
MESSAGES 52
MMAC_NDEVICES_QUERY 53
Purpose: 53

Versions: 53
Sample Usage: 53
lParam1: 53
lParam2: 53
MMAC_CAPS_QUERY 54
Purpose: 54
lParam2: 55
Description: 55
MMAC_DEVICEN_SET 56
Purpose: 56
Versions: 56
Sample Usage: 56
lParam1: 56
lParam2: 56
Description: 56
MMAC_o100_SRC_CONTROL 57
Purpose: 57
Versions: 57
Sample Usage: 57
lParam1: 57
lParam2: 58
Command Formats: 58
MMAC_o100_CC_CONTROL 62
Purpose: 62
Versions: 62
Sample Usage: 62
lParam1: 62
lParam2: 63
Description: 63
MMAC_o100_CC_FLAGS_GET 65
Purpose: 65
Versions: 65
Sample Usage: 65
lParam1: 65
lParam2: 65
Description: 65
MMAC_o100_CC_CONTROL2 67
Purpose: 67
Versions: 67
lParam1: 67
lParam2: 67
Command Formats: 67
Timestamps 71
Raw Mode Format 71
Line Mode Format 72
Screen Pointer and CC Mode 73
Relation of MMAC_o100_CC_CONTROL2 to MMAC_o100_CC_CONTROL 74
MMAC_o100_VBI_CONFIG 75
Purpose: 75
Versions: 75
lParam1: Points to an initialized VBIFMT structure, declared as follows in "o100ext.h": 75
lParam2: 76
MMAC_o100_VBI_CONTROL 77
Purpose: 77
Versions: 77
Notes: 77

MMAC_o100_VBI_ADD_BUFFER 78
Purpose: 78
Versions: 78
lParam2: 79
MMAC_o100_LOGO_CONTROL 80
Purpose: 80
Versions: 80
lParam1: 80
lParam2: 80
MMAC_o100_CROP_CONTROL 83
Purpose: 83
Versions: 83
lParam1: 83
lParam2: 83
MMAC_o100_REGISTER_GET 85
Purpose: 85
Versions: 85
Sample Usage: 85
lParam1: 85
lParam2: 85
Description: 85
MMAC_o100_REGISTER_SET 87
Purpose: 87
Versions: 87
Sample Usage: 87
lParam1: 87
lParam2: 87
Description: 87
MMAC_ADV_ CONTROL 89
Purpose: 89
Sample Usage: 89
lParam1: 89
lParam2: 90
Description: 90
MMAC_o100_SETTINGS_GET / SET 91
Purpose: 91
Versions: 91
Sample Usage: 91
lParam1: 91
lParam2: 91
MMAC_o100_SETTINGS_GET2 / SET2 92
Purpose: 92
Versions: 92
Sample Usage: 92
lParam1: 92
lParam2: 93
Description: 93
STRUCTURES 94

INTRODUCTION

This SDK allows applications to control the Osprey Video Capture Driver in ways that the Video for Windows (VfW) control interface cannot directly handle. It supplements VfW but does not replace it. The SDK bypasses VfW and sends messages directly to the video capture driver, via the multimedia commands OpenDriver(), CloseDriver(), and SendDriverMessage(). The following diagram shows the relation of the SDK to your application, to VfW, and to the Video Capture Driver:



The SDK works under Windows XP, 2000 and NT4.0. It works with Osprey-50, -100, -101, -2x0, -500, and -2000 series cards. Except where noted, all features work with driver release 2.0 and later for all Osprey cards.

The SDK addresses the following main areas:

- Control of multiple video capture boards by the same application.
- Control by the application of input source, input video format, brightness, contrast, hue, and saturation.
- Capture of Closed Caption text into the application.
- Capture of raw Vertical Blanking Interval (VBI) data into the application. (Osprey 100, 2x0 only)
- Control of an on-video logo from the application.
- Cropping of video images

- Selecting audio inputs
- Direct read and write of the Bt848 / Bt878 registers.
- Control of advanced video processing features

Improvements of SDK Version 2.1.0 Over Previous Versions

2.1 is the sixth release version of the kit – the first was designated 1.40, the second was 1.50, the third was 1.52, the fourth was 1.54 and the fifth was 2.0.

This information is for users of SDK version 2.1:

1. The `MMAC_o100_SRC_CONTROL` message supports the color correction bypass feature available on the Osprey-500 and Osprey-2000 cards. Use the `VIDEOCTLS` command to query or change the bypass setting.
2. The `MMAC_o500_CONTROL` message from the Osprey-500 SDK has been incorporated into this SDK and renamed `MMAC_ADV_CONTROL`. The `O500CTL` structure used by `MMAC_o500_CONTROL` has been renamed to `OADVCTL`. See the description of `MMAC_ADV_CONTROL` for more details.
3. A new example application, advApp, has been added to the SDK. This application demonstrates the use of the MMAC_ADV_CONTROL message.

This information is for users of SDK version 2.0:

1. Two new applications, vfwMixer and RegAdmin, have been added to the SDK. The vfwMixer application demonstrates how to control the audio settings for Osprey devices using the Video For Windows Mixer API. RegAdmin is a utility to aid system administrators in setting up and managing user and default settings for Osprey drivers. It can be used to set the registry of a system directly, or it can generate scripts that can be run on a facility-wide basis.

This information is for users of SDK version 1.54:

1. A new message, MMAC_o100_CROP_CONTROL, allows control of the cropping features added to release 1.54 of the driver. A new application, CropApp, has been added to the SDK to demonstrate.

This information is for users of SDK version 1.52 and earlier:

1. The `MMAC_o100_SRC_CONTROL` setting has been expanded to support setting Square or CCIR601 pixel aspect ratio.

This information is for users of SDK version 1.50 and earlier:

1. The `OpenDriver(…, PO100_OPEN)` interface is augmented to allow foolproof mapping of `OpenDriver()` and `capDriverConnect()` to the same device. Samples are consolidated and revised to use the improved interface.

This information is for users of SDK version 1.40 and earlier:

1. A new message, `MMAC_o100_CC_CONTROL2`, is added that provides greatly expanded Closed Caption capabilities. New capabilities include timestamps, multiple concurrent CC streams, CC streams to a process separate from the primary Video for Windows process, and direct access to the formatted screen data.
2. Three new messages, MMAC_o100_VBI_CONFIG, MMAC_o100_VBI_CONTROL, and MMAC_o100_VBI_ADD_BUFFER, allow stream-oriented capture of raw Vertical Blanking Interval (VBI) data into your application.
3. A new message, `MMAC_o100_LOGO_CONTROL`, allows direct control of an on-video logo from the application.

This information is for users of SDK versions prior to 1.40:

1. A new message, `MMAC_o100_SRC_CONTROL`, improves control of the video source. The old method using the `EXTSETTINGS` or `EXTSETTINGS2` structure will continue to be supported by the driver. The old method should not be used for new work, and developers are advised to convert existing applications to use the new functions. If you do not switch to the new method you will have an increasingly hard time mapping the video inputs of Osprey-100 board variants including new types that we will begin selling shortly. The new message handles video input mapping in a hardware-independent manner. Use of the new functions (as well as the new boards) does require that the underlying Osprey-100 video capture driver be updated to version 1.35 or later.
2. You can control the video source using a separate control applet. This allows you to use custom video source controls with standard or third-party applications for which you do not have source code. This feature requires version 1.40 or greater of the Osprey-100 Video Capture Driver.
3. The method of connecting to the driver via `OpenDriver()` is augmented. A new structure is passed to `OpenDriver()` that configures the connection and replaces two subsequent messages that used to be required.
4. A registry-based option for controlling access to multiple boards is added.
5. New messages are added to directly read and write the Bt848 / Bt878.

This SDK package documents SDK messages for 1.32 and earlier versions, but does not include sample code for the old messages. We will supply SDK version 1.32 on request to anyone needing this information.

Support and Feedback

Unless you have purchased phone support, support is by email only.

Email questions and comments to support@osprey.viewcast.com. Reference “Osprey VFW SDK” clearly in your email.

When requesting SDK support, please make sure your question is specifically about the SDK, not a general question about Video for Windows. Video for Windows is documented in detail in the Visual C++ or MSDN online docs - "Platform SDK -- Graphics and Multimedia".

If, however, you do identify a defect in our SDK, driver, or documentation, or want a feature added to the SDK, we want to hear about it. Please contact us.

Background: the Bt848/878

The Osprey-100, 200, 500 and 2000 class cards are based on the Bt848 / Bt878 / Fusion 878A single-chip video capture device. Conexant, the former semiconductor business of Rockwell, manufactures these devices. The device was originally manufactured by BrookTree, which was acquired by Rockwell. You can get chip-level documentation from www.conexant.com.

The Bt878 and Fusion 878A devices are a superset of the Bt848 that include an audio section. This SDK primarily covers the video section of the Bt878, which is nearly identical to the video section of the Bt848. Access to the audio section of the Bt878 is available through standard audio interfaces available in Microsoft Windows.

To Use This Kit You Need…

1. Windows NT 4.0, Windows 2000 or Windows XP.
2. An Osprey-50, -100, -101, -200, -500, or 2000 card.
3. Version 2.0.0 or later of the Osprey-100/200/220 NT/Win2000/XP Video Capture Driver, or version 2.1.0 of the Osprey-500 and Osprey-2000 drivers. Earlier versions can be used but do not support all of the SDK's capabilities. If you do not have the latest version, get it from ftp.viewcast.com/pub/OSP-100/winnt/latest or ../beta - if you are using Windows 2000 or an Osprey-2000, just navigate down to your operating system and card type from ftp://ftp.viewcast.com/pub/.
4. The Users' Guide for your Osprey card. It is included in the driver setup image as a .pdf file.
5. Microsoft Visual C++ v6.0. The sample code should work unmodified on earlier compilers, but you will have to recreate the .dsp or .mdp file manually
6. Familiarity with Video for Windows. Our email support does not include familiarization with VfW. On-line documentation for VfW may be found in the MSDN documentation that comes with the Microsoft Visual C++ package.
7. (OPTIONAL) The users' guide for the Conexant Fusion 878A, obtainable at www.conexant.com.
8. For advanced Closed Caption work, Closed Caption reference documentation. We do not supply this documentation.

Basic Techniques

The method of communicating with the driver is basically the same for all the special functions in this SDK. It is message-based and uses the multimedia functions `OpenDriver()`, `CloseDriver()`, and `SendDriverMessage()`. These functions are described in the Visual C++ documentation under *Platform SDK – Graphics and Multimedia Services – Multimedia Reference*.

Here is a sample of how the Osprey VFW SDK uses these functions. This is "quick start" information. For complete information, refer to the reference chapter and the code samples. In this particular sample, the application directly reads a Bt848 register to determine whether a video signal is present at the video input:

```
///////////////////////////////////////////////////////////////
//
#include "mmacdrv.h"
#include "o100ext.h"

///////////////////////////////////////////////////////////////
//
// VideoPresent():
// 1. Open the driver directly.
// 2. Determine whether video is present at the input.
// 3. Close the driver.
// Return:
// -  0 if not present
// -  1 if present
// - -1 if error
//
// Driver to use for your Osprey Card Type
#define DRIVER_NAME (L"o100vc.dll")   // for Osprey 1xx and 2xx Cards
// #define DRIVER_NAME (L"o500vc.dll")   for Osprey 5xx Cards
// #define DRIVER_NAME (L"o2kvc.dll")    for Osprey 2000 Cards

DWORD dwDevice;

int VideoPresent()
{
  // Open the capture driver directly, go around VFW:
  O100_OPEN oOpen;

  oOpen.dwSizeof = sizeof(O100_OPEN);
  oOpen.dwSig    = MMAC_INPUT_SIG;
  oOpen.dwFlags  = VID_CHANNEL;
  oOpen.dwDevice = dwDevice;

  HDRVR hDrvr = OpenDriver(DRIVER_NAME, NULL, (LPARAM)&oOpen);
  if (hDrvr == NULL)
    return -1;

  // Make sure the driver supports v1.52 commands:
  if ((oOpen.dwCaps & o100_CAPS_V152) < o100_CAPS_V152)
    return -1;

  // Read the bt848/878 Device Status Register, bit 7:
  REGISTER_CTL oRegCtl;

  oRegCtl.dwReg = 0;  // the address of this register

  BOOL bOk = SendDriverMessage(
    hDrvr,
    MMAC_o100_REGISTER_GET,
    (LONG)&oRegCtl,
    sizeof(REGISTER_CTL)
  );

  CloseDriver(hDrvr, 0, 0);

  if (!bOk)
    return -1;
```

```
  if (oRegCtl.dwData & 0x80)
    return 1;
  else
    return 0;
}
```

COMPONENTS OF THE SDK

The SDK has the following main components:

1. Accessing multiple boards
2. Controlling the video source
3. Controlling the video source from a separate process
4. Accessing Closed Caption data – preferred method – three samples
5. Accessing Closed Caption data – old method
6. VBI raw capture (1x0, and 2x0 cards only)
7. Controlling audio inputs
8. Accessing the Bt848 registers
9. Administration of registry settings for multiple users
10. Controlling advanced features

The general principles for each of the SDK components are described below. Each of the components has one or more sample applications associated with it. The samples are all in C++ and use Microsoft Foundation Classes and Microsoft Visual C++ 6.0. Some of the samples are single document interface applets; the rest are dialog based.

Accessing Multiple boards

This section explains different methods of accessing multiple Osprey cards in a system. It discusses how different windows operating systems handle access to video drivers. The App2Ctl example program is used to demonstrated the various access options.

The information in this section is focused on releases 1.52 and later of the driver and SDK. The version 1.40 SDK documentation gives addition information on how to interface to pre-1.52 versions of the driver.

The standard Video for Windows (VfW) method for connecting to a video capture device is to call `capDriverConnect(hCapWnd, dwIndex)`. `dwIndex` is a number from 0 to 9 corresponding to a video capture device.

The SDK uses a lower-level routine to connect to the driver, `OpenDriver(`*`drivername`*`, NULL, lParam)`. *`Drivername`* can either be the name of the driver DLL, `L"o100vc.dll"`, or the name of the form `L"msvideo"`, `L"msvideo1"`, `L"msvideo2"`, … that refers to a variable in *`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Drivers32`*. lParam is a pointer to either the VfW generic VIDEOPARMS structure or Osprey's proprietary O100_OPEN structure.

Note that the name of the dll is different for different Osprey Card types. The dll for Osprey 100 and 200 series cards is L"o100vc.dll". For Osprey 500 series cards the dll is L"o500vc.dll". For Osprey 2000 cards the dll is L"o2kvc.dll". You must use the correct driver name for the card type you wish to access.

If *`drivername`* is `L"o100vc.dll"`, and there are multiple Osprey-100-type devices, there is no inherent way to indicate which device is to be accessed. There is a private convention between the application and the driver – for the Osprey driver, the device number is embedded in a structure passed via `lParam`, as will be explained in detail.

If *`drivername`* is `L"msvideo#"`, the device number may be implied by the content of the registry string value referenced by `msvideo#`. Again, though, the system does not clearly define the device identification method, and it is a driver-specific convention.

When you use the Osprey SDK, you typically access the video capture device using both `capDriverConnect()` and `OpenDriver()` at once. The problem is that it is not automatic that the two opens will refer to the same device.

Under Windows NT 4.0 there is a simple one-to-one correspondence between the 0..9 `dwIndex` parameter to `capDriverConnect()`and the 0..9 `L"msvideo#"` parameter to `OpenDriver()`.

Under Windows 2000 and XP, however, a whole new enumeration method is added for Plug and Play devices. The `dwIndex` parameter to `capDriverConnect()` refers to a very particular enumeration of devices listed in the registry, and it does not correspond at all to the `OpenDriver(L"msvideo#", …)` convention.

Especially under Windows 2000/XP, therefore, problems can quickly arise if there are multiple Osprey-100/-200 cards, or a mix of these with Osprey-500 cards, or a mix of these with cards from other vendors.

Later subsections in this chapter describe registry-based access to VfW devices under the two operating systems.

Starting with version 1.52 of the driver, Osprey provides enhanced mapping methods to head off these problems. Basically, the steps are

1. Call `OpenDriver()` in any of three ways described below, referencing an `O100_OPEN` structure in `lParam`. Depending on the calling method, you will be either passing a `capDriverConnect()`-compatible `dwIndex` parameter to the driver, or getting one back from the driver.
2. Call `capDriverConnect()` with the `dwIndex` passed to or obtained from the `OpenDriver()` call.

All three options for `OpenDriver()` use the `O100_OPEN` structure. That structure is defined as follows:

```
typedef struct {
```

```
    DWORD dwSizeof;
    DWORD dwSig;
    DWORD dwDevice;
    DWORD dwFlags;
    DWORD dwCaps;
    DWORD dwError;
  }
  O100_OPEN, *PO100_OPEN;
```

The data format of the `dwDevice` member is what distinguishes which access option is being attempted.

Here are the details of the three options:

Option 1:

Call `OpenDriver()` with the first parameter set to the name of the driver dll. For Osprey-100/-200 cards the correct name is `L"o100vc.dll"`. For Osprey 500 cards use L"o500vc.dll" and for Osprey 2000 cards use L"o2kvc.dll".

On input to the `OpenDriver()` call, set `dwDevice` to the Osprey-100/-200 board instance that you want to access.

On output, the bits of `dwDevice` are as follows:

- Lower word (bits 0..15): The index of this device to be used in a subsequent `capDriverConnect()` call.
- Upper word (bits 16..31): Identification of the type of Osprey board being accessed. Bits 16..23 give the basic device category, as follows:

```
#define DVC_o100       0x00020000  /* Osprey-50/100/101/200  */
#define DVC_o500       0x00030000  /* Osprey-500             */
#define DVC_o2K        0x00040000  /* Osprey-2000            */
#define DVC_OTHER      0x00010000  /* none of the above      */
```

  Bits 24..31 are specific to each device category. For the Osprey-100/-200 these bits are 0.

Option 2:

Call `OpenDriver()` with the first parameter the name of the driver dll. For Osprey-100/-200 cards the correct name is `L"o100vc.dll"`. For Osprey 500 cards use L"o500vc.dll" and for Osprey 2000 cards use L"o2kvc.dll".

On input to the `OpenDriver()` call, set bits 0..15 of `dwDevice` to 0xFFFF. Set bits 16..31 to the `capDriverConnect()`-compatible index of the device to which you are connecting.

On output, the lower word of `dwDevice` (bits 0..15) contains the zero-based Osprey-100/-200 board instance being accessed. The upper word contains device type information as in Option 1.

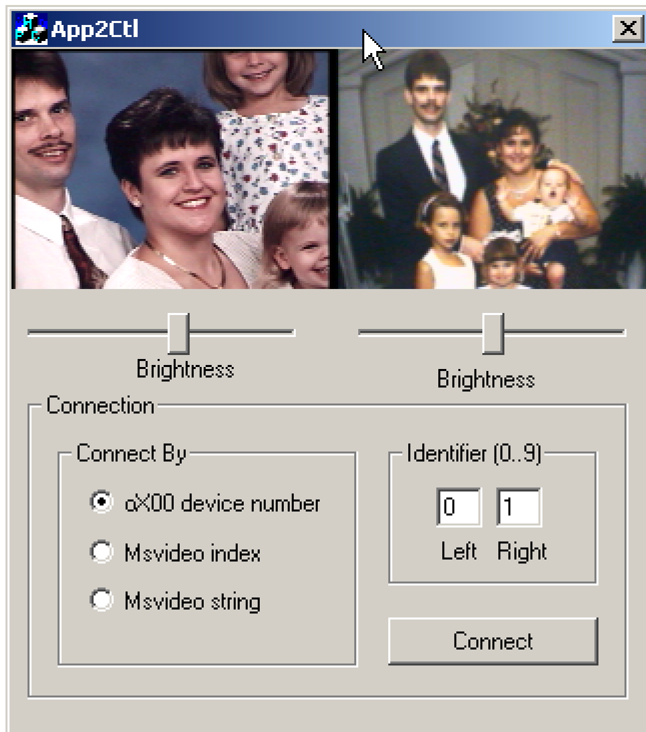
Option 3:

Call `OpenDriver()` with the first parameter set to `L"msvideo"` or `L"msvideo1"` through `L"msvideo9"`.

On input to the `OpenDriver()` call, set `dwDevice` to ~0 (0xFFFFFFFF).

On output, `dwDevice` is as in Option 1, with a `capDriverConnect()`-compatible index in the lower word, and device type information in the upper word.

App2Ctl:



The sample app App2Ctl illustrates these techniques. For each of two Osprey-100s it opens an SDK channel followed by a VfW channel. The VfW channel is used for overlay video. The app's window has two sliders that control brightness via the SDK channels. Three buttons select the three board access mode to use. Two edit boxes set the access parameters for each card, which could be their Osprey device numbers, capDriverConnect() indexes, or msvideo registry entries.

App2Ctl.dsp, like all the sample project files, is for VC++ 6.0. If you are running an earlier version of the compiler, create a skeleton MFC app with the following characteristics:

- project name APP2CTL
- dialog-based
- files App2Ctl.cpp / App2Ctl.h and CtlDlg.cpp / CtlDlg.h
- dialog class CCtlDlg
- link with vfw32.lib and winmm.lib

Then overwrite the wizard-created .cpp and .h files with the ones provided in this kit.

Background: msvideo Registry Keys – Windows NT 4.0:

Under Windows NT 4.0 all msvideo keys are located in

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Drivers32

There can be up to 10 keys. Key 0 is named *msvideo*, the others are named *msvideo1..msvideo9*.

These keys have string values. The first part of the string is the name of the DLL that should be started when `OpenDriver()` is called with this key – o100vc.dll in the case of the Osprey-100/-200, o500vc.dll for the Osprey-500, and o2kvc.dll for the Osprey-2000. The optional remainder of the string is passed to the video capture driver. The Osprey drivers look at the last character of the string. If it is a digit from 1 to 9, that digit is interpreted as the one-based number of a device belonging to that driver that the application is trying to access.

For example, a complex case with five boards might look like this:

```
msvideo:      o100vc.dll – Osprey Video Capture Card 1
msvideo1:     o500vc.dll – Osprey-500 Card 1
msvideo2:     o500vc.dll – Osprey-500 Card 2
msvideo5:     brandx.dll
msvideo6:     o100vc.dll – Osprey Video Capture Card 2
```

Under NT 4.0 there is simple direct mapping between the VfW index and *msvideo#*. A call to `capGetDriverDescription(0, …)` will return data for the first of these items (*msvideo*), `capGetDriverDescription(1, …)` will return data for *msvideo1*, and so forth. There are no *msvideo3* and *msvideo4*, and `capGetDriverDescription(3..4, ..)` will return FALSE. As expected, `capGetDriverDescription(5..6, ..)` will return data for *msvideo5* and *msvideo6*.

Your application may be expecting a simple layout such as,

```
msvideo:      o100vc.dll – Osprey Video Capture Card 1
msvideo1:     o100vc.dll – Osprey Video Capture Card 2
```

You might, for example, be expecting `OpenDriver(L"msvideo", …)` to reliably open the first of two Osprey-100 cards and `OpenDriver(L"msvideo1", …)` to open the second card. Clearly, this is a limiting assumption about how *msvideo* keys and devices might be set up. It could lead to SDK / VfW mis-mappings if other devices are added, and could unduly restrict your clients' configurations. We therefore recommend following one of the three new `OpenDriver()` options. More specifically, the first two options are usually better since they get away from the *msvideo#* keying entirely.

One minor point to note is that some applications, including earlier versions of VidCap32, expect the *msvideos* to be numbered contiguously. These earlier VidCap32s will not enumerate msvideo5 and msvideo6 from the above example in its menu list of available drivers. Other applications take an even simpler approach and look only at *msvideo[0]*.

Background: msvideo Registry Keys – Windows 2000/XP:

Under Window 2000/XP, there are two perturbations to the straightforward mapping described above. First, Plug and Play devices are added that are registered elsewhere from the …*\Drivers32 msvideo* keys. Second, the index numbering of *msvideo* entries is changed.

Consider again the example:

```
msvideo:      o100vc.dll – Osprey Video Capture Card 1
msvideo1:     o500vc.dll – Osprey-500 Card 1
msvideo2:     o500vc.dll – Osprey-500 Card 2
msvideo5:     brandx.dll
msvideo6:     o100vc.dll – Osprey Video Capture Card 2
```

`capGetDriverDescription()` under Windows 2000/XP will return TRUE with valid data when called with indexes 0 through 4. `capGetDriverDescription(4, …)` will return the data for *msvideo6*. In other words, there is no direct mapping between enumeration indexes and *msvideo* numbers, except that they will be in the same order.

Consider now a further example:

```
MSVideo.VFWWDM:
              Vfwwdm32.dll
              Microsoft WDM Image Capture (Win32)
msvideo:      o100vc.dll – Osprey Video Capture Card 1
```

```
msvideo1:      o500vc.dll – Osprey-500 Card 1
msvideo2:      o500vc.dll – Osprey-500 Card 2
msvideo5:      brandx.dll
msvideo6:      o100vc.dll – Osprey Video Capture Card 2
MSVIDEO8:      vfwwdm32.dll
```

The new *MSVideo.VFWWDM* entry is located not in `...\Drivers32` but in

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\MediaResources\msvideo

Msvideo.VFWWDM is a key rather than a value, and contains the string entries shown for *Driver* and *Description*, as well as other information. `capGetDriverDescription(0, …)` will return the WDM entry, and `CapGetDriverDescription(1..5, …)` will return the five …\Drivers32 entries.

MSVIDEO8 is a shadow to *MSVideo.VFWWDM*. The WDM driver install process puts it there so that applications that are not *MediaResources*-aware can find it. It is not, however, enumerated by `capGetDriverDescription()`.

The Windows 2000/XP enumeration always orders the *MediaResources* entries first, followed by the *Drivers32* entries. The *MediaResources* entries are in ASCII order. The method of listing capture devices does not include the concept of a "default device" being enumerated as the first item. First, there is no way to order the *MediaResources* items other than by renaming them, and we do not know at present whether this can be done without side effects. Second, since *MediaResources* items are always enumerated first, there is no straightforward way to force a *Drivers32\msvideo* entry to be the first device enumerated. This means that VfW client applications with expectations of a particular default device will be disappointed.

As a workaround, the Osprey driver's Configuration dialog Default Capture Device control under Windows 2000/XP will create a *MediaResources* key named *!default* (the '*!*' forces it to sort first) if requested to make a *Drivers32\msvideo* driver the default driver. The 1.52 driver has this behavior, but not the 1.51 version. The *MediaResources\msvideo\Driver* entry is a duplicate of the *Drivers32\msvideo* entry.

Specific Osprey Driver Behavior – Hunt Mode:

The "hunt" capability is an older method of accessing multiple Osprey devices. It may still be useful in some instances. Its main ongoing value is that it works well with applications that only know how to access the default device, that is, the device enumerated by *capGetDriverDescription(0, …)*.

In a nutshell, hunt mode works as follows: The driver keeps a number for a default device as part of its private registry settings. When hunt mode access is made to the driver, the driver tries to open the default device for a specified purpose, such as video or control access, as identified by flag parameters in the low-level open call. If the device is already opened for that purpose, whether by the current process or some other process, the driver then searches through its other devices in order starting with device 0 and returns a handle to the first available device if any. It follows that, in hunt mode, in the general case you cannot open the same device for a given purpose multiple times. You can have multiple handles to the device only if they are for different purposes – for example a VFW handle and an SDK handle.

In non-hunt mode, there is no default device – you must always directly or indirectly tell the driver which device you are trying to open. You can open the same device multiple times for any purpose, and you will get back multiple handles to it. This is the behavior that most of this chapter has focused on.

Driver versions prior to 1.51 always had hunt mode as their primary access mode. Driver versions starting with 1.51 retain that behavior, but only as an alternative access mode that must be explicitly enabled by the Hunt Mode checkbox on the Configuration page of the driver's control dialog - their default

behavior is no-hunt mode. This change was necessary because major applications are requiring access to the same video stream from very disparate components.

To use hunt mode:

1. Hunt mode must be enabled. It is always enabled with pre-1.51 drivers but must be explicitly turned on in 1.51 and later drivers, by checking Hunt Mode on the Configuration control page.
2. The open parameter received by the driver, whether from a direct `OpenDriver()` call or indirectly from a `capDriverConnect()` call, must be a pointer to an `LPVIDEOPARMS` structure. If you call capDriverConnect(), VfW always creates and passes the required `LPVIDEOPARMS` structure to the driver.
3. The open call to the driver must not explicitly or implicitly reference a *…\Drivers32\msvideo* string that ends in a digit. The driver interprets such a digit as a target device number when the open driver call is made with an `LPVIDEOPARMS` parameter, and overrides the hunt behavior.

The registry requirement in item 3 is usually not an issue with pre-1.51 drivers – driver installation sets up a single msvideo entry for the driver, with the undecorated string value “o100vc.dll”. You would have to have manually changed the registry to have a problem. Starting with driver 1.51, you have to make sure that you access the msvideo string with the value “o100vc.dll - Osprey Capture Card Default” that is set up by the driver when it is in hunt mode.

Specific Osprey Driver Behavior – Setting up Registry Keys:

Versions of the Osprey-100/-200 driver prior to version 1.51 do not perform registry manipulations. The installer sets up a single *msvideo* entry with the string value *"o100vc.dll"*. No provision is made for automatically setting up *msvideo* strings that reference specific devices.

Versions 1.51 and later of the driver sets up *msvideo* registry keys for its boards each time it starts, and removes them each time it stops.

If hunt mode is not enabled – the default – the keys are similar to the following:

msvideo: *o100vc.dll – Osprey Video Capture Card 1*

msvideo1: *o100vc.dll – Osprey Video Capture Card 2*

If hunt mode is enabled the keys are similar to the following:

Msvideo: *o100vc.dll – Osprey Video Capture Card Default*

Msvideo1: *o100vc.dll – Osprey Video Capture Card 1*

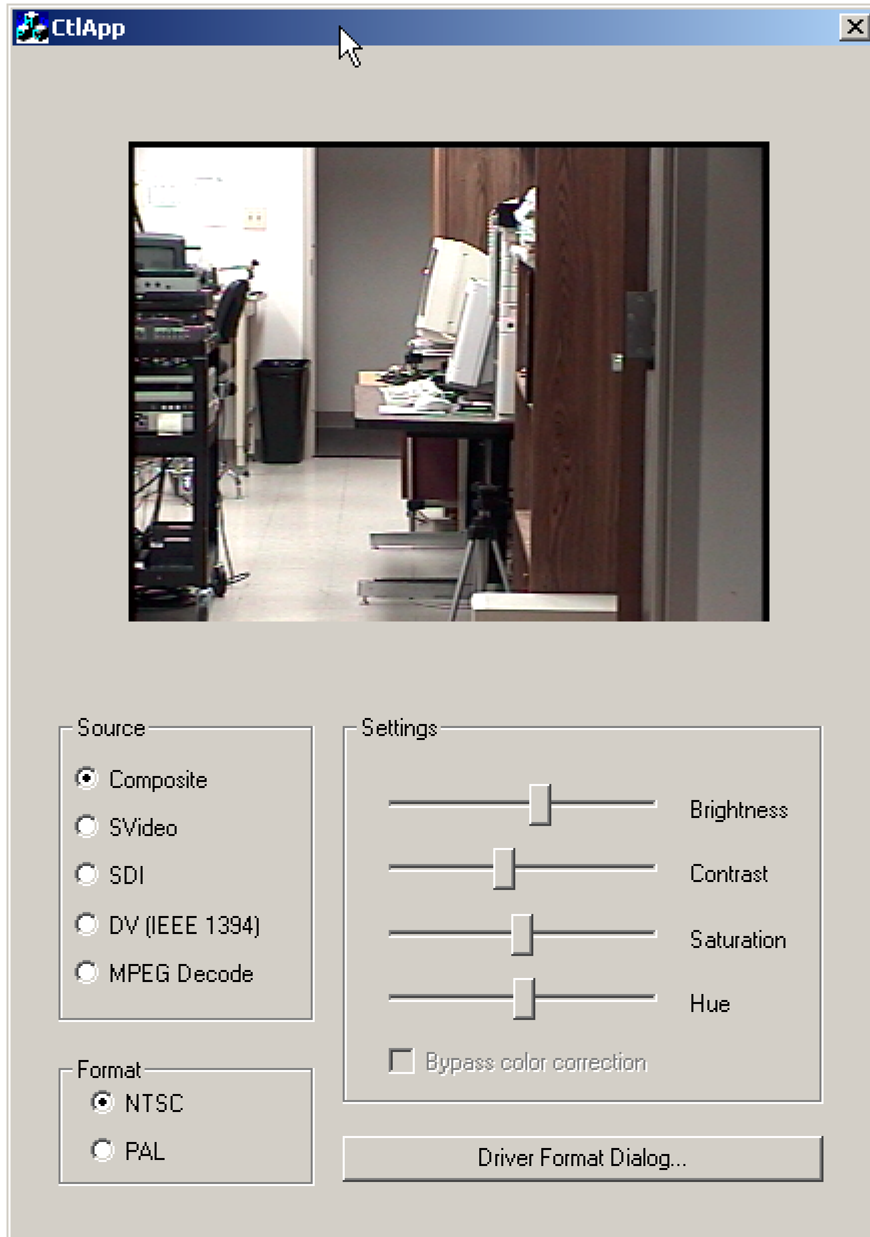
Msvideo2: *o100vc.dll – Osprey Video Capture Card 2*

If *msvideo* entries for other drivers are present, the driver will search the *msvideo* entries in order from 0 to 9 and use the first available entries.

If the Osprey-100/-200 driver is told by its configuration dialog that it is the default driver it will put its first entry in the *msvideo[0]* slot if that slot is otherwise unused. If it is not set as the default driver it will not use *msvideo[0]* even if it is vacant. To turn on the non-default behavior, use the driver’s Configuration dialog page, Default Capture Device control, to select a default device that is not an Osprey-100/-200.

As previously stated, under Windows 2000/XP, the default device entry, if it is an Osprey-100/-200/-500, is placed under a *…\MediaResources\msvideo\!default* key as well as *…\Drivers32\msvideo*.

Controlling the Video Source - CtlApp



Note: Input choices will depend on your actual board type.

This section explains how to Control the video source settings on Osprey cards. The CtlApp sample application is used to demonstrate how to get the current settings and how to change them. It displays the video above the controls used to change the setting so that you can see the effect of your changes.

The video source functions that can be controlled include

1. The video input connector.
2. The video format (NTSC, PAL, SECAM).

3. Brightness, contrast, saturation, and hue.

CtlApp.dsp is for VC++ 6.0. If you are running an earlier version of the compiler, create a skeleton MFC app with the following characteristics:

- project name CTLAPP
- dialog-based
- files CtlApp.cpp / CtlApp.h and CtlDlg.cpp / CtlDlg.h
- dialog class CCtlDlg
- link with vfw32.lib and winmm.lib

Then overwrite the wizard-created files other than CtlApp.dsp with the ones provided in this kit.

CtlApp's substantive SDK-related code is in CTLDLG.CPP. Important routines include the following:

- `CCtlDlg::OnShowWindow()` – establishes a connection to the Osprey-100 driver.
- `CCtlDlg::OnDestroy()` – shuts down the connection.
- `CCtlDlg::Enumerate()` – enumerates the board's video inputs.
- `CCtlDlg::SetSource()` – sets the video source.
- `CCtlDlg::SetVFmt()` – sets the NTSC/PAL/SECAM video format.
- `CCtlDlg::OnHScroll()` – sets brightness, contrast, saturation, hue.

Of these routines the most complex is CCtlDlg::Enumerate(). Osprey sells a number of board types with different type, numbers, and arrangements of video inputs. The enumeration function determines how many of each input type – composite and svideo – are present. It also determines what video formats – NTSC, PAL b/d/g/h/i/m/n, and SECAM – the board supports.

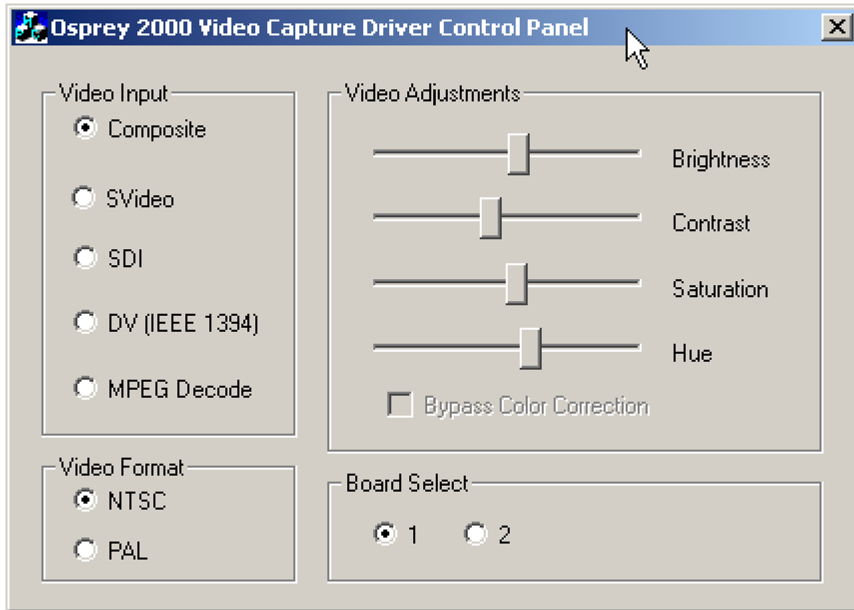
CCtlDlg::Enumerate() and the last three routines in the list use the complex SDK message MMAC_o100_SRC_CONTROL. This message has five main command formats, described in detail in the Messages reference section. These subcommands allow the message to do the following:

- Get information about the available video input sources.
- Set the video input source.
- Get information about available input video formats.
- Set the video input format.
- Get and set brightness, contrast, saturation and hue.

All getting and setting of video parameters is done in terms of logical rather than physical input source settings. An application that correctly enumerates the input sources will have its control messages mapped to the correct video input regardless of the physical arrangement of connectors on the board.

Although the Bt848/878 has a single set of registers to control brightness, contrast, saturation and hue, the Video Capture Driver maintains separate, logically mapped settings for each input.

Controlling the Video Source from a Separate Process - XCtlApp



Note: Video Input choices will depend on the actual Osprey Card type.

This section explains how to Control the video source settings on Osprey cards. The XCtlApp sample application is used to demonstrate how to get the current settings and how to change them. It does not display the video so you must use a separate application such as vidcap to see the effect of any changes you make to the settings.

This sample app is the same Control Panel that is included with the Osprey-100 retail package starting with version 1.50.

Whereas the "CtlApp" described previously is a complete standalone application combining a VfW overlay screen with control functions from the SDK, "XCtlApp" is an auxiliary control-only application that has SDK functions but no VfW functions. It accesses the same Osprey board as another application – for example a standard app that does not provide adequate video source control.

XCtlApp also illustrates how to control the video inputs of multiple boards. NOTE: The board select buttons do *not* change which board the primary application is accessing. They are for use when multiple primary apps are running, or when a single primary app is accessing multiple boards at once. They determine which primary process XCtlApp is connected to and is controlling.

Since XCtlApp is derived from CtlApp the class and file names are the same as CtlApp's. CtlApp.dsp is for VC++ 6.0. If you are running an earlier version of the compiler, create a skeleton MFC app with the following characteristics:

- project name CTLAPP
- dialog-based
- files CtlApp.cpp / CtlApp.h and CtlDlg.cpp / CtlDlg.h
- dialog class CCtlDlg
- link with vfw32.lib and winmm.lib

Then overwrite the wizard-created files other than CtlApp.dsp with the ones provided in this kit.

The key routines are in CtlDlg.cpp and are the following:

- `CCtlDlg::OnShowWindow()` – multiboard-aware initialization.
- `CCtlDlg::CtlOpen()` – open a board and enumerate its inputs.
- `CCtlDlg::OnDestroy()` – shuts down the connection.
- `CCtlDlg::Enumerate()` – enumerates the board's video inputs.
- `CCtlDlg::SetSource()` – sets the video source.
- `CCtlDlg::SetVFmt()` – sets the NTSC/PAL/SECAM video format.
- `CCtlDlg::OnHScroll()` – sets brightness, contrast, saturation, hue.

Many routines in XCtlApp are similar to CtlApp – in particular, Enumerate(), SetSource(), SetVFmt(), and OnHScroll(). Two routines – `OnShowWindow()` and `CtlOpen()` – are different. The key difference is how the `dwFlags` member of `O100_OPEN` is set:

```
/* CtlApp:  */  oOpen.dwFlags = VID_CHANNEL;
/* XCtlApp: */  oOpen.dwFlags = CTL_CHANNEL | CTL_LOCK;
```

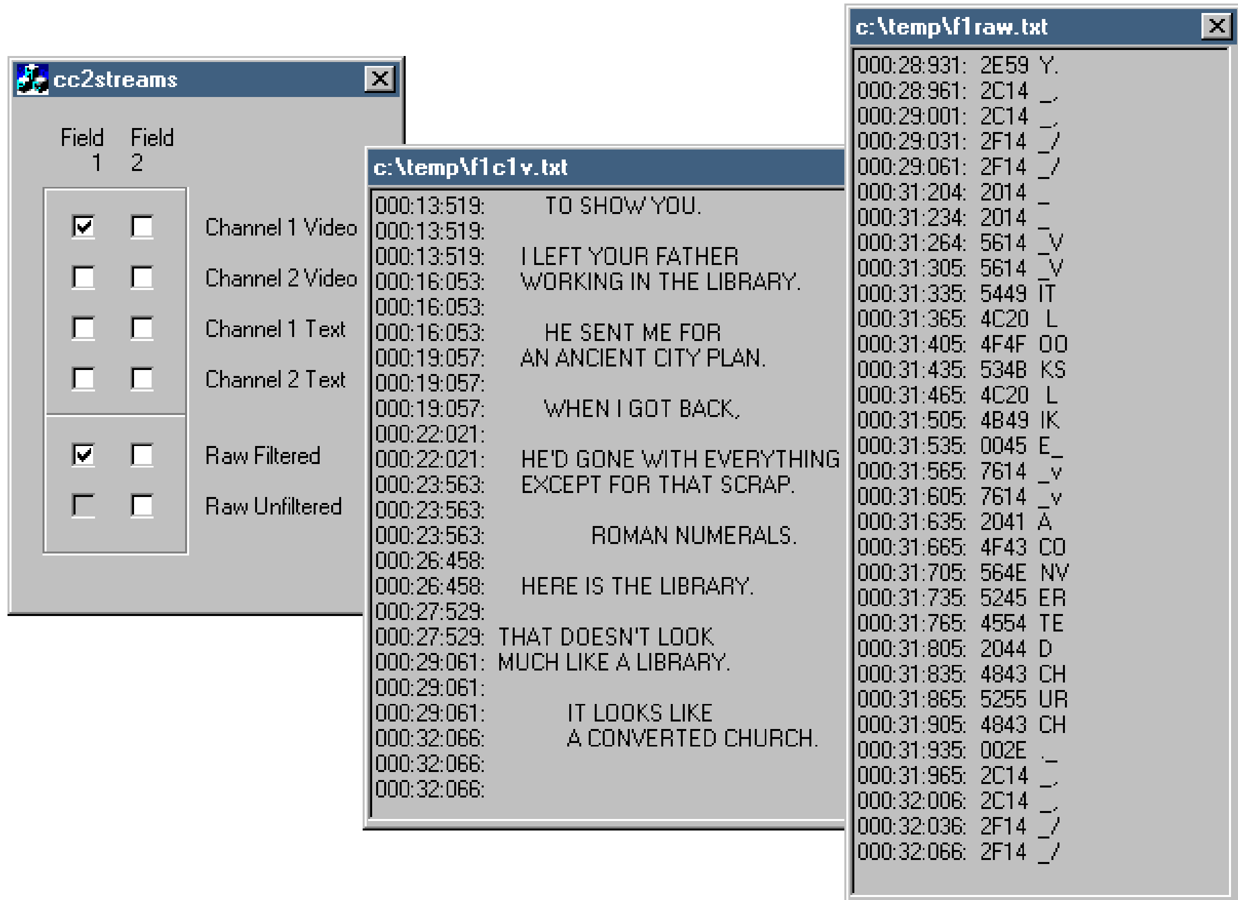
When `VID_CHANNEL` is set, the app is asking for exclusive access to the designated Osprey-100 board. If another application attempts to access that board, that access will fail, whether it is through VfW or directly through `DriverOpen()`.

When `CTL_CHANNEL` is set, the app is asking for non-exclusive access to control functions only. Another application can access the board for VfW or other video access.

When `CTL_LOCK` is set in addition to `CTL_CHANNEL`, the app is directing the driver to lock out the other application from making video source control adjustments. It is generally recommended to set this flag.

The `CTL_CHANNEL` mode is only supported by driver versions 1.40 and higher.

Accessing Closed Caption Data – “Generation-2” Method - CC2Streams



This component of the SDK allows you to control Closed Caption (CC) operations in the driver from your app, and to obtain CC data from the driver. “Closed Caption” is a standard used in North American NTSC broadcasting for transmitting character data with each video frame. The first active line of video in the odd or even field is replaced by a waveform containing a character pair.

Cc2streams demos the preferred generation-2 `MMAC_o100_CC_CONTROL2` command interface described in detail later in this document.

The application shows how to open simultaneous streams, with or without a Video for Windows application simultaneously accessing the same board from a separate concurrent process. CC text can be captured in a decoded line-oriented format for any of eight channels. It can also be captured as raw character pairs, with or without filtering for data validity, for field 1 and/or field 2. The main application window has twelve checkboxes that allow you to open windows for the possible flavors of CC data. The data appearing in the windows is simultaneously captured to the files named in the window titles.

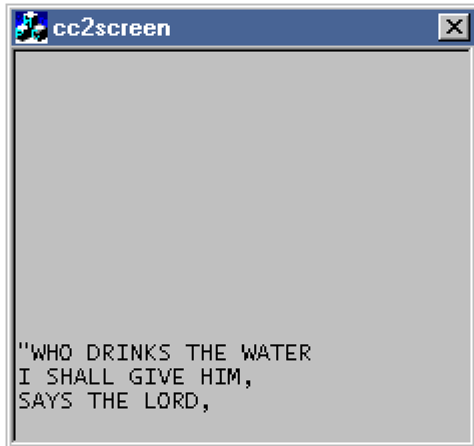
Cc2streams is organized as a dialog based application. Overall control of the CC windows resides in the files ccDlg2.cpp and ccDlg2.h. Line mode windows are defined by class `CLine` in line.cpp and line.h. Raw mode windows are defined by class `CRaw` in raw.cpp and raw.h. The raw mode class is derived from the line mode class.

This demo uses text edit controls for text display. This results in easy-to-understand sample code, but it should be noted that it is not very efficient in a production application.

Note the use of the `PostMessage()` call in line.cpp and raw.cpp. This call is only necessary in applications that write text to a screen window. It decouples screen output from the driver's callback thread. This has been found necessary to prevent application lockups under some circumstances.

This is a Visual C++ 6.0 project. If you are using a different compiler you can create your own skeleton and fold in the substantive sample code.

Accessing Closed Caption Data – “Generation-2” Method - CC2Screen



Cc2screen demos the ability of the generation-2 Closed Caption interface to provide direct access to the decoded, formatted screen data. It uses the `MMAC_o100_CC_CONTROL2` command interface described in detail later in this document. This interface allow you to access CC data from a board at the same time that a video window with CC enabled is running in either the same process or in another process; multiple copies of cc2screen or similar applications can access the same board simultaneously.

The application consists of a dialog box with a single edit control sized for 15 rows by 32 columns of non-proportional text. This dimensioning corresponds to the size of the Closed Caption screen area.

Cc2screen is hardwired to board 0 – though your application does not have to be. It is hardwired to open a line mode channel to field 1, channel 1, video mode - the channel for most CC text – again for the sake of simplicity. The unused line mode buffer pointer, `CC_CALLBACK_CTL2.pcData`, is specified as NULL to slightly reduce driver overhead.

At each callback, the screen's ASCII data is moved to an edit-box-formatted character buffer and copied to the edit box. The current screen pointer is specified at each callback in `PCC_CALLBACK2->pwScreen`. Note that, in pop-on CC mode, the buffer address this variable points to will change each time a new caption is popped on.

In each screen data word, the ASCII character codes are in the lower 7 bits of each word; the upper word contains attribute information.

There is one ASCII code that most apps will have to specially interpret: Non-CC blank space is represented as ASCII NULLs (0x00) in the lower 7 bits. In this example, leading and in-CC-text blank space is converted to ASCII space characters, and trailing blank space is truncated.

This simple example does not try to display the per-character attributes in the upper byte of each screen word. The data format is straightforward, however, and you could fairly easily augment this applet to show text with colors, underlines, italics, and winking. The following #defines are from <o100ext.h>:

```
#define CC_CHAR      0x00FF  /* ascii character field  */
#define CC_COLOR     0x0700  /* color field            */
#define CC_WHITE     0x0000  /* the colors             */
#define CC_GREEN     0x0100
#define CC_BLUE      0x0200
#define CC_CYAN      0x0300
#define CC_RED       0x0400
#define CC_YELLOW    0x0500
#define CC_MAGENTA   0x0600
#define CC_UNDERLN   0x0800  /* underlined             */
#define CC_ITALICS   0x1000  /* italicized             */
#define CC_ATTR_MARK 0x2000  /* used internally by driver, don't change */
```

```
#define CC_FLASH     0x4000  /* flashing character     */
```

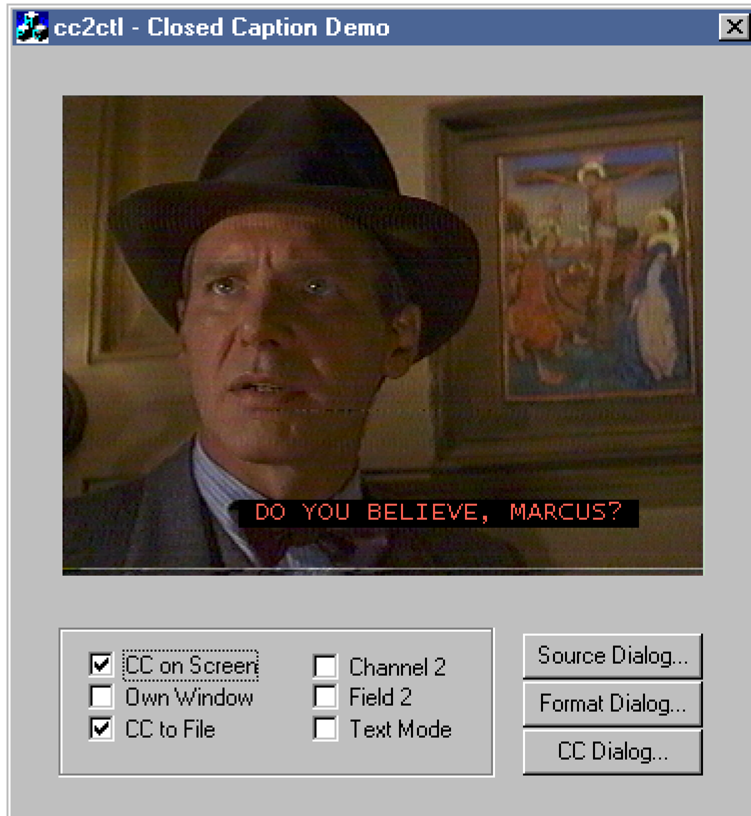
For example the code 0x1658 represents a magenta, italicized capital 'X'.

This demo uses a text edit control for text display. This results in easy-to-understand sample code, but it should be noted that it is not very efficient in a production application.

Note the use of the `PostMessage()` call in line.cpp and raw.cpp. This call is only necessary in applications that write text to a screen window. It decouples screen output from the driver's callback thread. This has been found necessary to prevent application lockups under some circumstances.

Cc2screen is organized as a dialog based application. It is a Visual C++ 6.0 project. If you are using a different compiler you can create your own skeleton app and fold in the substantive sample code.

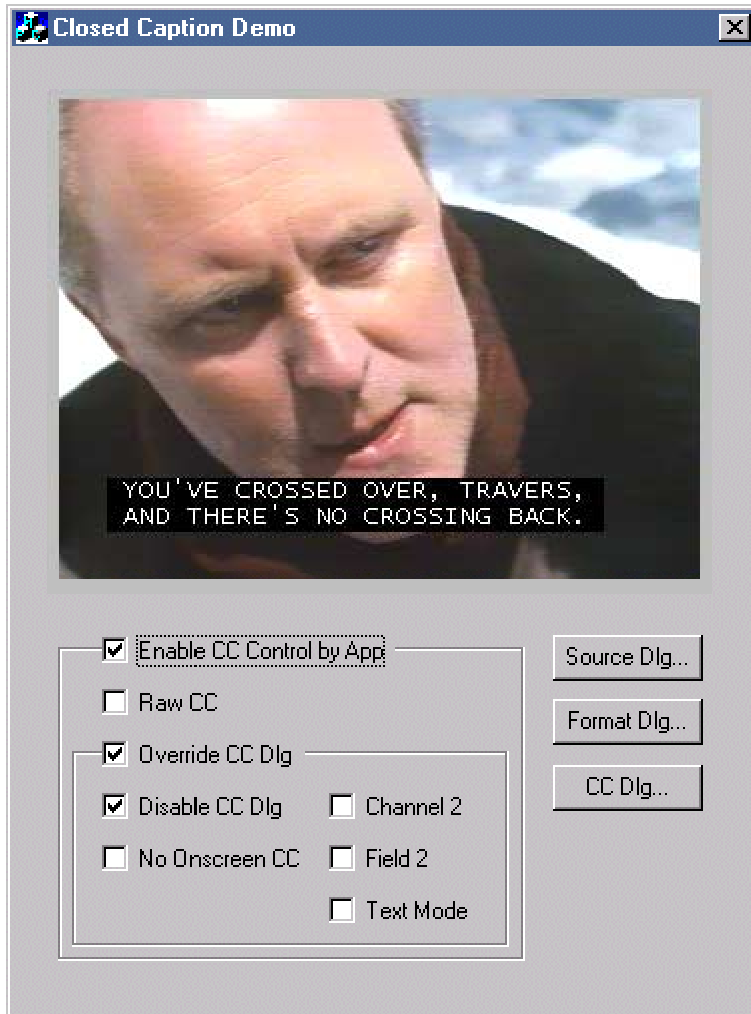
Accessing Closed Caption Data – "Generation-2" Method - CC2Ctl



This application illustrates the ability of the "generation-2" `MMAC_o100_CC_CONTROL2` interface to control the driver's internal CC screen display. For example, it lets you enable and disable on-screen display of CC independent of the user control dialog settings.

Sample App

Cc2ctl is a modified and simplified version of CcApp, the original CC control demo.Cc2ctl is a dialog-based applet with control buttons by which you can select CC options and control capture of CC text to the application's file. The applet does not have the callback features in CcApp because these are illustrated in the preceding to samples, Cc2streams and Cc2screen. The remarks about general program structure in the CcApp discussion above also apply to Cc2Ctl. The most important illustrative routines are `CCcDlg::OnShowWindow()` and `CCcDlg::Cc2Cmd()`.



Accessing Closed Caption Data – Old Method - CcApp

Note: This application demonstrates the SDK's original and less capable CC interface, governed by the messages `MMAC_o100_CC_CONTROL` and `MMAC_o100_CC_FLAGS_GET`. The enhanced `MMAC_o100_CC_CONTROL2` interface is recommended for new applications that do not need to support driver versions earlier than 1.50. The original interface will continue to be supported, however.

The `MMAC_o100_CC_CONTROL` CC interface lets you override the user-accessible CC controls in the Osprey-100 Video Capture Driver's control dialog's Closed Caption tab. For example, you can enable and disable on-screen display of CC independent of the user control dialog settings.

In addition, the CC interface allows your application to capture CC text. You can capture the text in either of two forms:

1. Lines of plain text – "line mode"
2. Raw CC character pairs – "raw mode"

Both are delivered by callbacks to your application.

The original `MMAC_o100_CC_CONTROL` interface requires that video overlay or capture be active in order for CC data to be returned to your application (This restriction is removed by the

`MMAC_o100_CC_CONTROL2` interface.). You do have the options of displaying or not displaying CC text on the overlaid or captured video.

In driver versions prior to 1.40, vertical video size should be 240 lines or more in order for CC to work reliably. Starting with driver version 1.40, the size of the video frame does not affect CC.

This kit does not explain what CC is, how it works, or (especially) how to interpret "raw mode" CC codes. You would have to get this information from standard reference sources. But you do not need this information to successfully utilize the plain text data delivered by line mode.

In line mode, the lines of data are NULL-terminated strings using the 256-character ANSI ASCII character set. Certain CC character codes that are not in the basic 128-character ASCII set are remapped to the extended ASCII set. The maximum length of a CC line is 32 bytes, so a line mode line buffer is 33 bytes in size. Line mode strips out color, italics, and underlining information. For the most part it preserves the horizontal position of characters by padding the beginnings of lines where required. Blocks of text are grouped as contiguous lines. A line is skipped wherever a new caption is popped on. A line skip or carriage return is signified by a callback with an empty line. Multiple empty lines in a row are suppressed.

In raw mode, all character pairs are delivered to your application except for pairs in which both characters are NULL.

Sample App

CcApp is a dialog-based applet with control buttons by which you can select CC options and control capture of CC text to the application's file.

The project file, CCApp.dsp, is in Microsoft VC++ 6.0 format. If you are using an earlier version of the compiler, create a skeleton app with the following characteristics:

- project name CCAPP
- dialog-based
- files CcApp.cpp / CcApp.h and CcDlg.cpp / CcDlg.h
- dialog class CCcDlg
- link with vfw32.lib and winmm.lib

Then overwrite the wizard-created files with the files from the SDK (except for CCApp.dsp).

Nearly all the substantive code is in CcDlg.cpp.

Look especially at `CCcDlg::OnShowWindow()` for how to connect to the driver, start video overlay, and create a control channel to the Video Capture Driver for CC operations.

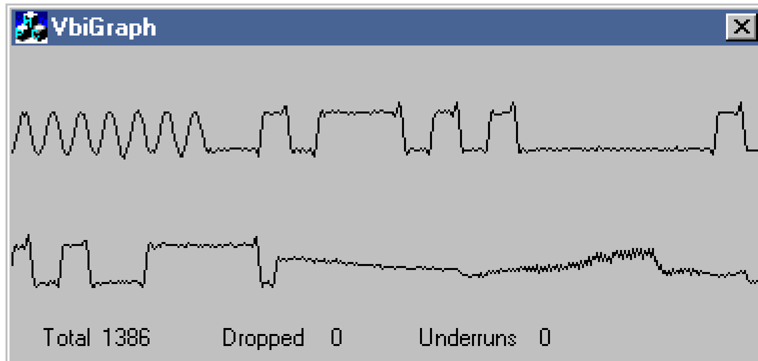
Look also at `CCcDlg::CcCtl()`, which sends CC control bits to the driver.

Control of CC uses the SDK message `MMAC_o100_CC_CONTROL` and its associated structure, `CC_CALLBACK_CTL`. These are described fully in the Messages reference under `MMAC_o100_CC_CONTROL`.

The two callback routines, `CCRaw()` and `CCLines()`, receive CC text from the driver and write it to file.

Your callback function is called by a capture thread belonging to the driver, and so should be fairly quick, by which we mean well under 1/60 of a second in duration. If more extensive processing is required, capture the data with the callback and process it fully with your main thread.

VBI Raw Capture - VbiGraph



Using version 1.50 and later of the Osprey-100 driver and SDK, an app can capture raw Vertical Blanking Interval (VBI) data from the driver. The Osprey-100 and 2x0 series cards support capturing the raw VBI data. The Osprey-500 and –2000 series cards do not support raw VBI capture.

VBI capture uses a streaming paradigm modeled on normal VfW video capture. The application allocates data buffers and queues them in the driver. The driver returns filled buffers to the app, and the app requeues them when it has decoded the data.

VBI capture can coexist with video capture, either in the same app or in a different app. For example, you should be able to run the sample app VbiGraph at the same time that you are displaying video from the same board with VidCap32. The one restriction to bear in mind is that each line of video can go to only one destination. That is, you cannot both capture a given line of video as VBI and capture it as video. And, you cannot capture the Closed Caption line both as decoded CC and as raw VBI. You can, however, capture the CC line as decoded CC and the VBI lines preceding it as raw VBI.

The sample app VbiGraph displays lines of raw VBI data graphically - one line of VBI from the odd field and one line from the even field. In the sample screen shown above, a line of NTSC Closed Caption data is shown along with the corresponding line of the even field.

VbiGraph reads the driver's current input video format setting – NTSC, PAL, etc., and adjusts the VBI capture parameters accordingly.

VbiGraph is a dialog-based applet. The app-specific code is in VbiGraphDlg.cpp. Look especially at the following routines:

- CVbiGraphDlg::VbiInit()
- Callback()
- CVbiGraphDlg::Graph()
- CVbiGraphDlg::OnGraph()

Set up VBI capture using the following steps, following VbiInit() as a model:

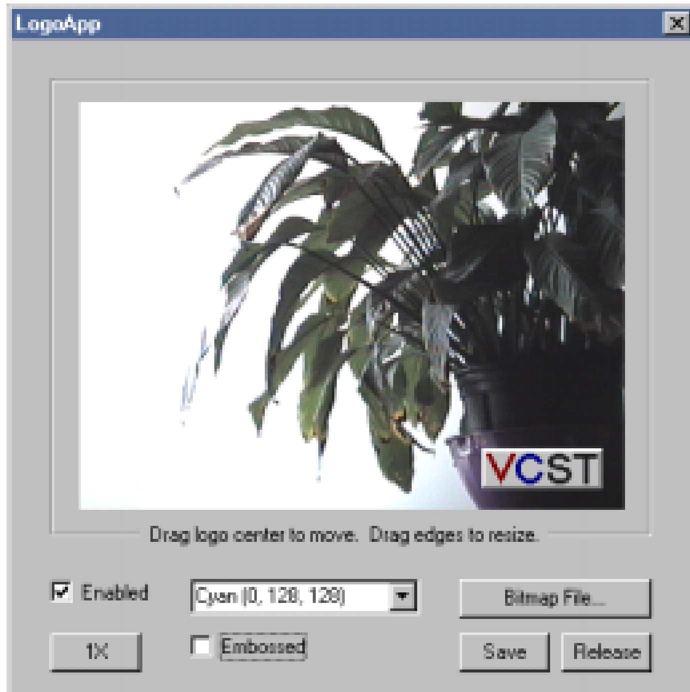
1. Open a VBI stream in the driver, obtain a handle to it.
2. Make sure the driver is 1.50 or later (recommended, not mandatory).
3. Detect the NTSC/PAL video format (or set your desired format).
4. Set the low-level parameters in the VBIFMT structure.
5. Pass the VBIFMT structure to the driver with a VBI configuration message.
6. Allocate the VBI buffers and queue them in the driver.
7. Start the VBI stream running.

The callback routine then graphs the returned VBI buffers. The VBI buffers are packed arrays of bytes. The callback routine also tracks number of frames, number of frames dropped by the driver, and number of underruns.

The VbiGraph callback has a two-layered structure. The outer layer uses PostMessage() (note – Post-, not SendMessage()) to decouple the substantive graphing routine from the driver's internal thread. This design averts a potential freezeup condition. It may not be required in applications where the callback thread does not directly send text or graphics to the application window.

As with the other sample apps in this kit, VbiGraph is a VC6 project. If you are using a different compiler you can create your own skeleton and fold in the substantive sample code.

Logo Control - LogoApp



LogoApp illustrates how to control the Osprey driver's logo capability from an application. Driver versions starting with 1.50 have the logo capability. The Video Capture Driver Users' Guide describes the general nature of this capability in detail. This description assumes that you are familiar with it.

LogoApp applet illustrates the following specific controls:

- Enabling and disabling the logo.
- Selecting the logo bitmap file.
- Selecting the keycolor and enabling/disabling "embossed" mode.
- Moving and resizing the logo.
- Saving changes to the registry.
- Releasing control of the logo back to the driver.

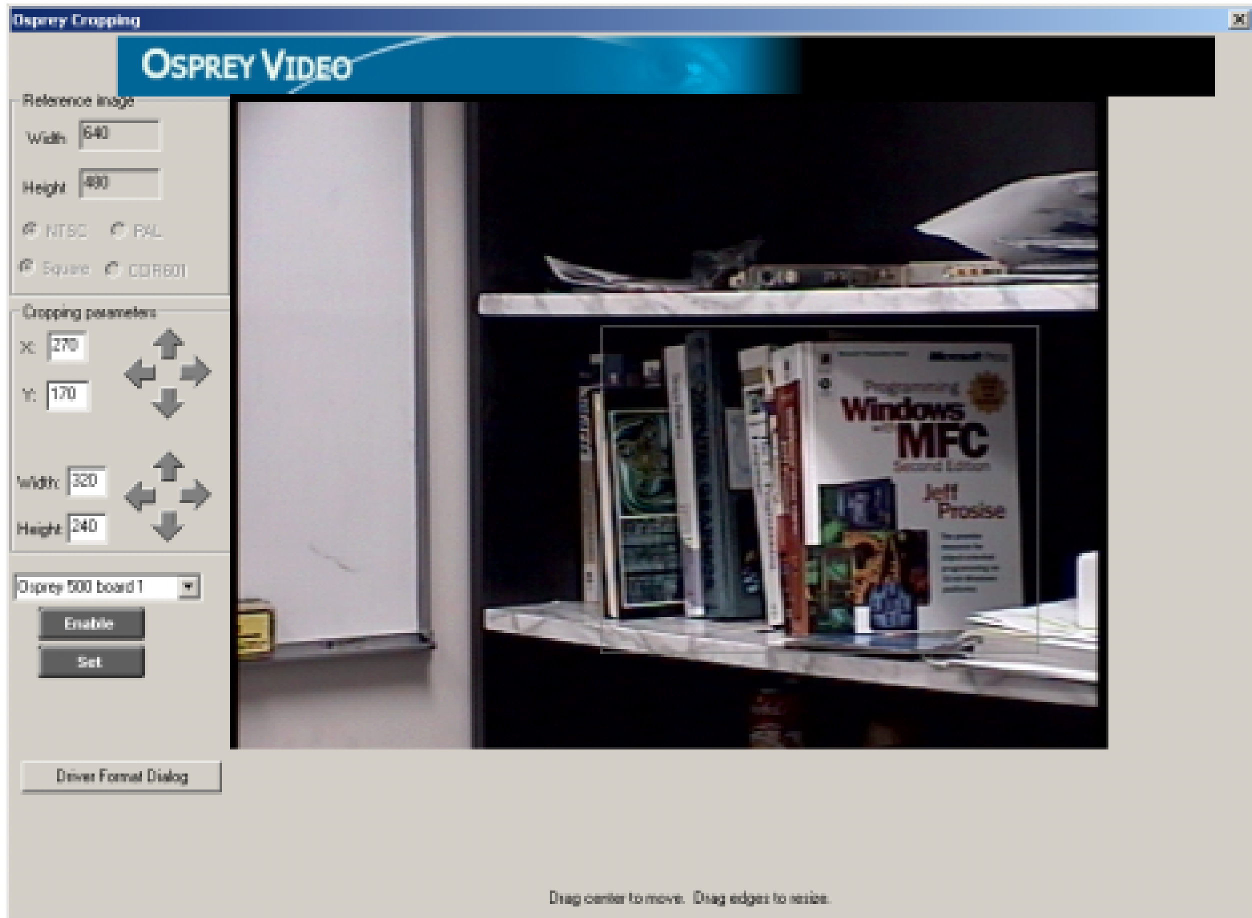
Note that you **must** be in **DibDraw** mode in order to view the logo with the LogoApp. LogoApp uses a single SDK message, `MMAC_o100_LOGO_CONTROL`, and its five subcommands to accomplish these functions. Refer to the detailed description of this message and the LOGO_CTL structure in the reference section of this document.

LogoApp is a Visual C++ 6.0 dialog-based application. The basic methods of connecting to and controlling the driver are similar to those used for the other sample applications. The differences pertain to allowing direct on-video moving and resizing of the logo. A separate window class, `CScale`, in LogoScale.cpp, is derived from `CWnd` and a single object of this class, `oScaleWnd`, is instantiated. Mouse events over the video are directed to the `oScaleWnd` instantiation of this class. The VfW capture window, `hCapWnd`, is then created as a child of oScaleWnd. It is created with the WS_DISABLED attribute. This attribute allows video to be visible and at the same time directs mouse events to the underlying `oScaleWnd`.

One point to note is the following: If a `LOGO_SET` message contains incorrect parameters, the driver may adjust them. For example, the driver does not allow a logo to be partially off the edge of the video field. If

the dwLogoTop or dwLogoLeft settings would result in that condition, they will be adjusted. You can follow a LOGO_SET message with a LOGO_GET message to determine if this has happened.

Cropping Control – CropApp



CropApp demonstrates how to control the Osprey driver's hardware cropping capability from an application. Driver versions starting with 1.54 have the cropping capability. The Cropping User's Manual describes the general nature of this capability in detail. This description assumes that you are familiar with it.

CropApp applet illustrates the following specific controls:

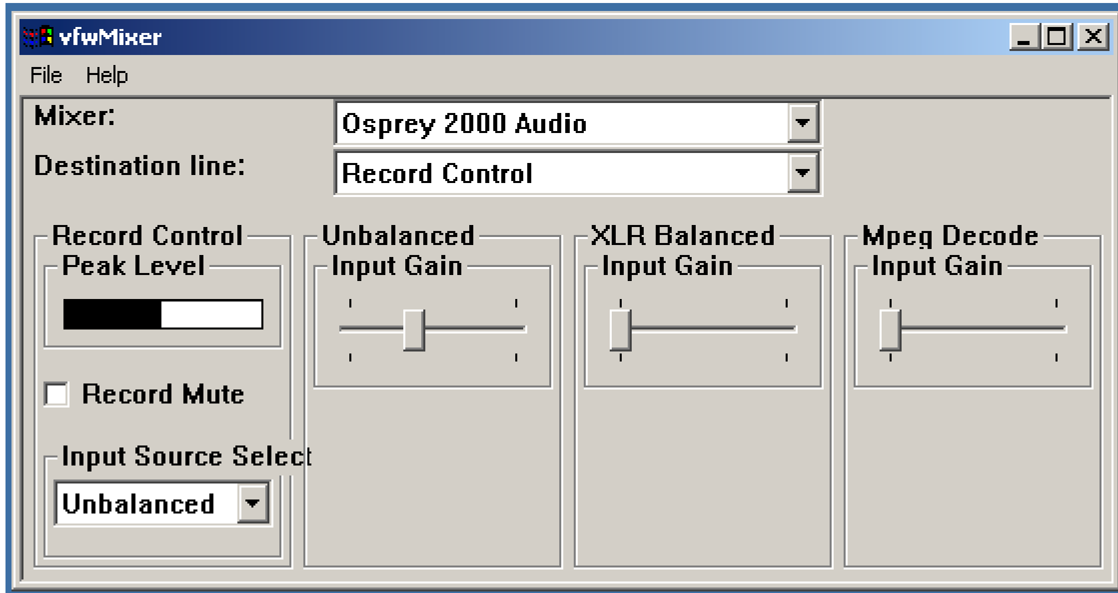
- Enabling and disabling cropping
- Selecting the cropping box
- Saving changes to the registry
- Setting and reading video format and pixel aspect ratio

CropApp uses a single SDK message, `MMAC_o100_CROP_CONTROL`, and its two subcommands to accomplish these functions. Refer to the detailed description of this message and the CROP_CTL structure in the reference section of this document.

CropApp is a Visual C++ 6.0 dialog-based application. The basic methods of connecting to and controlling the driver are similar to those used for the other sample applications.

The CROP_SET command may alter the parameters passed to it based on hardware limitations. Not only may the cropping not exceed the original image size, but the cropping parameters may also be rounded down to the nearest value that the hardware will support.

Controlling Audio Settings - vfwMixer



The vfwMixer application demonstrates how to control the audio settings for Osprey devices using the Video For Windows Mixer API. It uses the VFW Mixer API to find out what audio devices are installed and what attributes are available for the different source and destination lines for the device. It displays a Graphic User Interface to allow the user to adjust the settings on the various devices.

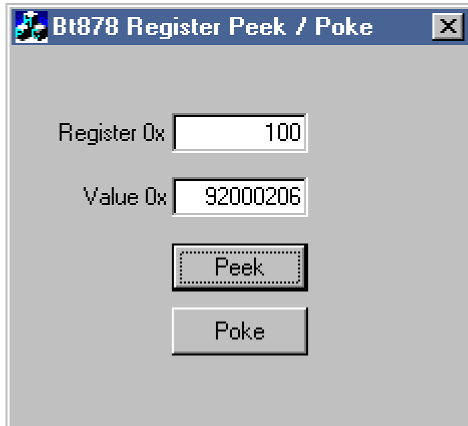
Each audio capable device in the system is represented as a Mixer. Mixers have one or more destination lines and each destination line has one or more source lines. Each source and destination line has one or more controls such as a volume control or a mute button depending on the device. This sample application can control audio on third party sound cards as well as Osprey devices.

The application is best used in combination with other tools that capture Audio/Video from an input device. For example you could use VidCap32 to capture Audio/Video with the vfwMixer application running to adjust the audio attributes since VidCap32 provides very little control over the audio.

VfwMixer uses the VFW Mixer API to find all the Mixer devices when it starts up. It does this by calling the mixerGetNumDevs() function to determine how many mixer devices are present and then calls the mixerGetDevCaps() function to get information about each device. This all takes place in the getMixerInfo method of the MixerInfo class. For more information on the VFW Mixer API consult the Microsoft documentation.

The application displays a CComboBox control to allow the user to select which Mixer device to control and another one to select which destination line to manipulate. It then displays all the controls for that destination line and the controls for all the source lines associated with the destination line. The current settings are reflected by the GUI controls and if the GUI is changed by the user then the application will make the corresponding changes to the settings for the mixer device using the VFW Mixer API function mixerSetControlDetails(). The ChildView::updateControlDisplay() method is executed periodically by a timer to detect changes that might be made to the device settings by other applications and the display is updated if any changes are found.

Accessing the Bt848/878 registers - RegApp



The project file, RegApp.dsp, is in Microsoft VC++ 6.0 format. If you are using an earlier version of the compiler, create a skeleton app with the following characteristics:

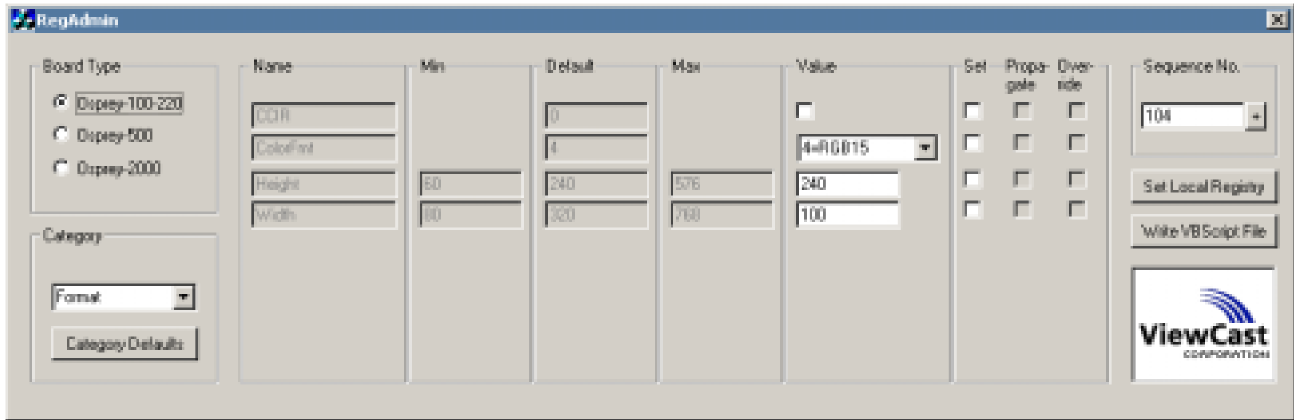
- project name REGAPP
- dialog-based
- files RegApp.cpp / RegApp.h and RegDlg.cpp / RegDlg.h
- dialog class CRegDlg
- link with vfw32.lib and winmm.lib

Then overwrite the wizard-created files with the files from the SDK (except for RegApp.dsp).

The SDK-related code is in RegDlg.cpp. Look at `CRegDlg::OnInitDialog()`, `CRegDlg::OnPeek()`, and `CRegDlg::OnPoke()`. The method is built on the SDK messages `MMAC_o100_REGISTER_GET` and `MMAC_o100_REGISTER_SET`, and their associated structure `REGISTER_CTL`. For complete information about these, refer to the Message references for `MMAC_o100_REGISTER_GET` and `MMAC_o100_REGISTER_SET`.

Use of these functions requires Conexant Bt848/878 documentation, not supplied with this kit, obtainable at www.conexant.com. Direct setting of registers in this way is generally not recommended. Osprey can generally not assist debug of applications that use `MMAC_o100_REGISTER_SET`. Note that these registers are runtime settings that are not necessarily persistent.

RegAdmin



RegAdmin is a utility to aid advanced users and system administrators in setting up and managing user and default settings for Osprey drivers. It can be used to set the registry of a system directly, or it can generate scripts that can be run on a facility-wide basis.

Windows 2000 and Windows XP support .WSF scripts. Windows NT does not support .WSF – use VBS scripts with Windows NT. If you have not installed Windows Scripting Host on your machine, the most recent version of Microsoft's Script can be downloaded from:
http://www.microsoft.com/msdownload/vbscript/scripting.asp

RegAdmin supports the following board types:

- Osprey-100, -200, -210, -220
- Osprey-500
- Osprey-2000

RegAdmin is intended mainly for use with the version 2.00 and later drivers. It can be used with partial functionality with earlier drivers.

These driver versions incorporate features that enhance administrators' ability to administer user settings on a facility-wide basis. The new capabilities include:

1. Ability to set custom defaults and to automatically propagate them to users' private settings.

2. Ability to override users' private settings. Although the user can change overridden settings in the course of the current session, these settings will not be saved to disk. Since some of the controls in the driver's control dialogs are configured from the registry, it is also possible to limit the choices that the user can access.

3. Some specific variable definitions are changed for device-independence and clarity.

RegAdmin Quick Tour #1

1. Start RegAdmin.

2. At the top left, select the type of board that you want to revise the default settings for.

3. Look at the drop box in the Category group – it defines several categories or pages of variables. The main portion of RegAdmin shows the variables in the category you have currently selected. To follow along a very benign example of changing the default settings, choose *Comp1* in the Category group drop box.

4. On the top data line, *Brightness*, there is an edit box that will initially show the value 5000. Change it to 2000. Note that the *Set* checkbox becomes checked.

5. Click *Set Local Registry*.

6. Run an app such as VidCap32 with an active source attached to the Composite 1. Select the Composite 1 input in the driver's control panel, Source page. Click *Restore Defaults*. The video should become quite dark and the *Brightness* slider should move to correspond to the default you have set.

7. To put brightness back the way it was, go back to RegAdmin, select the Comp1 page again if necessary, and click *Category Defaults*. It should now show the default value of 5000. Again click *Set Local Registry*.

RegAdmin Quick Tour #2

1. Run VidCap32 to verify that Brightness is at a normal level. Exit VidCap32.

2. Start RegAdmin.

3. At the top left, select the type of board that you want to revise the default settings for.

4. Choose *Comp1* in the Category group drop box.

5. On the top data line, change the value in the *Brightness* edit box again to 2000. The *Set* checkbox becomes checked.

6. On the *Brightness* line, this time also check the *Propagate* box.

7. Click *Set Local Registry*.

8. Run VidCap32 again with Composite 1 selected and an active source attached to that input. Without any adjustment of the driver's controls, the video should be quite dark and the *Brightness* slider should correspond to the 2000 value you set in RegAdmin.

9. In the VidCap32 source control dialog, set the Brightness slider to a bright setting and click OK. Exit and restart VidCap32 and note the new bright setting is retained.

10. To put brightness back the way it was, go back to RegAdmin, select the Comp1 page again if necessary, and click *Category Defaults*. Again *click Set Local Registry*.

RegAdmin Quick Tour #3

1. Run VidCap32 to verify that Brightness is at a normal level. Exit VidCap32.

2. Start RegAdmin.

3. At the top left, select the type of board that you want to revise the default settings for.

4. Choose *Comp1* in the Category group drop box.

5. On the top data line, change the value in the *Brightness* edit box again to 2000. The *Set* checkbox becomes checked.

6. On the *Brightness* line, uncheck the *Propagate* box and check the *Override* box.

7. Click *Set Local Registry*.

8. Run VidCap32 again with Composite 1 selected and an active source attached to that input. Without any adjustment of the driver's controls, the video should be quite dark and the *Brightness* slider should correspond to the 2000 value you set in RegAdmin.

9. In the VidCap32 source control dialog, set the Brightness slider to a bright setting and click OK. Exit and restart VidCap32. Note that this time the new bright setting was *not* retained.

10. To restore the normal default, go back to RegAdmin, select the Comp1 page again if necessary, and click *Category Defaults*. Again *click Set Local Registry*.

RegAdmin Quick Tour #4

1. Choose one of the first three scenarios and follow it up to the point of clicking *Set Local Registry*.

2. Instead click Write VBScript File and choose a filename as requested. RegAdmin will append a .vbs filename extension.

3. Open My Computer and navigate to the script file. Double click on it. The results should be the same as those obtained from writing to the registry directly. You can run the script on multiple machines. Unfortunately, the "propagate" function if used only works the first time the script is run on any given machine.

A Note on Sequence Numbers

At the upper right of RegAdmin there is an edit box entitled Sequence No., and a [+] button beside it. If you were watching carefully in Quick Tour #2, you would have noted that this number incremented when you clicked *Set Local Registry*.

Sequence numbers are used to make the "notify-each-user-once" scenario work. Normally, RegAdmin automatically increments the sequence number as needed.

You might need to adjust sequence numbers manually if you are preparing a script that will run on multiple machines on which the sequence numbers are variant or uncertain.

Since the sequence number range is from `0 to 4,294,967,294` (`0xFFFFFFFE`), you can increment it lavishly without fear of running out of space.

Specific Registry Variables

Format Category: `CCIR`, `ColorFmt`, `Height`, and `Width`: All these variables should be self-explanatory, or nearly so. The `CCIR` flag works as follows: If `CCIR` is cleared, the maximum or reference video width is `768` for PAL/SECAM, and `640` for NTSC. If `CCIR` is set, the maximum or reference video width is `720` for both the PAL/SECAM and NTSC standards.

Source Category:

1. `Input` refers to the source connections provided on the board. The available connections are different for each board type, and the drop list is updated whenever you change the Board Type selection. For the Osprey-100, -200, -210, and -220, the RegAdmin has provision for up to three composite inputs; the -200, -210, and -220 hardware only implements one of these possible inputs.

2. `Standard` refers to the format of analog input video – NTSC, PAL, or SECAM. The possible choices and the numeric values representing them are spelled out in the drop list. Only the formats shown in the driver's source dialog box are actually supported by the driver; the others are possible at the hardware level but are currently experimental only.

Config Category:

1. `UseDDraw` is a user-settable variable can take on the following meaningful values:

   0 – disable DirectDraw.
   2 – enable DirectDraw.

   Some of the other bits have undocumented and unsupported uses; we recommend setting only these two values.

2. `VFmtDlgCfg` controls which video standards (NTSC, PAL_BDGHI, etc.) are visible and selectable on the source page of the driver's control dialog. The lower seven nibbles of this DWORD correspond to seven control positions on the control dialog's source page. The numeric code in each nibble corresponds to the video standard that is to appear at that position in the dialog. Code `0` results in a blank space in that position. The other standards are encoded as shown on the Standard line on RegAdmin's source page (NTSC is `1`, PAL_BDGHI is `3`). For example, if you set `VFmtDlgCfg` to `0x00000001`, NTSC is the only choice that can be selected. If you set `VFmtDlgCfg` to `0x00000003`, PAL_BDGHI is the only choice that can be selected. You would at the same time set `Standard` on the source page to NTSC, and set its "propagate" flag.

3. `VModes` controls which video color formats (RGB32, YUY2, etc.) the user can select from in the drop list in the control dialog. You do so by using RegAdmin to set the configuration variable. Bits 1 through 12 of `VModes` enable or disable color formats 1 through 12. The color format numbering is shown on RegAdmin's format page. The default value is `0x18DE`. If you change that to `0x18D0`, RGB32 and RGB15 are effectively disabled. You would also want to set `ColorFmt` on the format page to one of the enabled values and set `ColorFmt`'s "propagate" flag.

`VFmtDlgCfg` and `VModes` are "configuration" variables rather than user-settable variables. They are placed in the same Osprey registry branch as the other defaults, but are not changeable by the user. Therefore, the "propagate" and "override" flags do not apply to them, and RegAdmin's checkboxes for these flags are always disabled.

Comp1, Comp2, Comp3, Svideo1, SDI1, DV1, Mpeg1: These categories all contain the same five variables: `Brightness`, `Contrast`, `Hue`, `SatU`, and `SatV`. `SatU` and `SatV` are the only ones that require special explanation: In the present driver implementation, these variables are redundant hardware parameters that the driver always keeps in fixed relation to each other. The equation is: `SatV = SatU * 0.709`.

The RegAdmin Source Code

RegAdmin is mainly intended as a pre-built tool rather than as a demo or code sample. While we provide source code, we have not extensively commented it, and do not plan to extensively support user modifications to it. RegAdmin communicates only with the registry, never directly with the driver, so none of the driver extensions used by the other applets are applicable. You will, however, find that it is largely table-driven, and it is fairly easy to add new variables into it as long as they are analogous to variables that are already there.

Background: The NT/2000/XP Registry

The Windows NT/2000/XP registry has the following contradictory characteristics:

1. Users without administrative privileges can write only to `HKEY_CURRENT_USER`.
2. Administrators cannot write to `HKEY_CURRENT_USER` keys other than their own.

It is therefore difficult to allow users to keep private settings while at the same time making possible for an administrator to propagate changes to all users.

The solution Osprey is using is to add to the registry what we are calling “sequence” information. The sequence information tells the driver when to use settings from `HKEY_CURRENT_USER`, when to use settings from `HKEY_USERS\.DEFAULT`, and when to update `HKEY_CURRENT_USER` with settings from `HKEY_USERS\.DEFAULT`. The RegAdmin utility handles many of the details of setting up the sequence keys.

Background: Basic Structure of the Osprey Registry Settings

Default settings for Osprey-100, -200, -210, and -220 devices are kept in

```
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100
```

Default settings for Osprey-500 and Osprey-2000 devices are respectively kept in

```
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey500
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey2000
```

Individual users’ settings are kept in

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100
HKEY_CURRENT_USER\Software\Osprey\Osprey500
HKEY_CURRENT_USER\Software\Osprey\Osprey2000
```

In the default tree, using the Osprey-100 as an example, settings for the video driver are all under

```
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\VcDll
```

Settings for brightness, contrast, hue, and saturation are kept individually for each input. They are placed under descriptive keys named Comp1, Comp2, Comp3, Svideo1, SDI1, DV1, and Mpeg1. These keys are all located under VcDll – for example:

```
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\VcDll\Comp1
```

Individual users’ settings are analogous, except that a Device# key is added to the path so that settings for individual devices can be kept individually. For example:

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\VcDll
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\VcDll\Comp1
```

When the driver wants to access a value from the registry, let's say `Width`, it first looks for

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\VcDll\Width
```

If this value is not found, it then looks in the default branch for

```
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\VcDll\Width
```

If this value is found, it is not only used in the current session, but is also written to

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\VcDll\Width
```

If the value is not found in the registry, the driver uses its own hard-coded internal default.

Background: The Sequence Keys

What is described so far is the basic registry model used by the Osprey-100/200/500/2000 drivers since their inception. What has been added starting with version 1.60 are the following new behaviors.

First, for each registry value there is an optional *sequence value*. Sequence values are DWORDs located in a separate "shadow" branch of the Osprey registry. For example, the `Width` and `Comp1\Hue` values and their associated sequence values are located as follows:

```
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\VcDll\Width
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\Sequence\VcDll\Width
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\VcDll\Comp1\Hue
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\Sequence\VcDll\Comp1\Hue
```

A big part of what RegAdmin does is to set up these sequence values.

As part of the sequence process, the driver writes its own sequence values in the user's registry tree. For example, the user's Width value and its associated sequence value would be at

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\VcDll\Width
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\Sequence\VcDll\Width
```

That is the structure. Now, this is what happens:

When the driver wants a value from the registry, let's say `Width`, it first reads two sequence values:

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\Sequence\VcDll\Width
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\Sequence\VcDll\Width
```

If the user's sequencing value is greater than or equal to the default sequencing value, then the driver reads its data from

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\VcDll\Width
```

If the user's sequencing value is less than the default sequencing value, then the driver instead uses the default value,

```
HKEY_USERS\.DEFAULT\Software\Osprey\Osprey100\VcDll\Width
```

In this case, it also writes the default `Width` and `Width`'s default sequencing value to

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\VcDll\Width
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\Sequence\VcDll\Width
```

The next time the driver accesses `Width`, it will then be able to directly read the user's value

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device0\VcDll\Width
```

The effect of the sequence values is to propagate "suggested" user settings from the default registry branch to each user's registry branch. The administrator needs only to set the defaults in one place, and is assured that all users will pick up these settings the next time they use the driver. It is assured that all users will see the suggested defaults; however, in this mode they are free to use them as is, or override them. If overridden, they will always be recoverable via the driver's "Restore Defaults" button.

There is another operating mode: Normally the sequence values are set to small, sequential numbers. However, if a sequence value is set to the largest possible value, `4,294,967,295`, `0xFFFFFFFF`, or `-1`, or `~0`, the effect is different from what was just described. In this case, the driver will never use the user's value; it will always use the default value. The user is still allowed to change the driver's internal working values for the duration of the current session, but these values are effectively volatile. When the driver is restarted, it will revert again to the default values.

Configuration Variables Vs. User Variables

Most of the Osprey registry variables are "user" variables that the user can by default override. A few variables are "configuration" variables that users cannot change. Configuration variables are stored under `HKEY_USERS\.DEFAULT` in the same way as other default variables, but do not propagate to `HKEY_CURRENT_USER`. In RegAdmin, the "Propagate" and "Override" flags are not meaningful, and are therefore always shown therein as disabled.

As of this writing, the only two variables exposed in RegAdmin in this category are `VModes` and `VFmtDlgCfg`.

Controlling Advanced Video Processing

The advanced video processing features on Osprey cards are controlled through the functions CCtlDlg::BoardSettings() and CCtlDlg::SetBoard(). Both functions use the new SDK message MMAC_ADV_CONTROL. The new video processing features are described below:

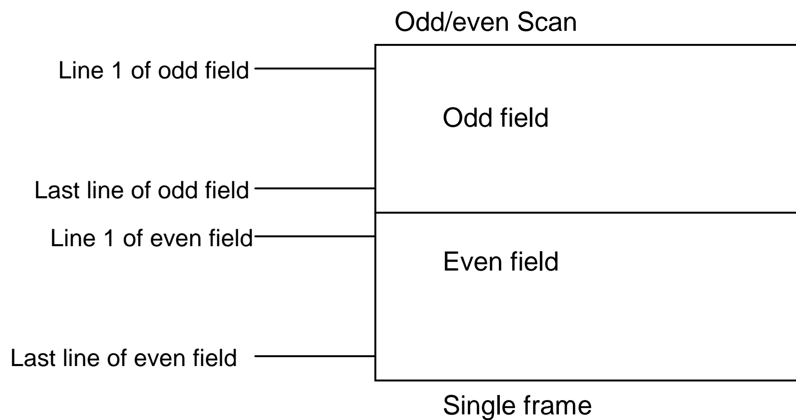
Analog Bypass (Osprey-500 and Osprey–2000)

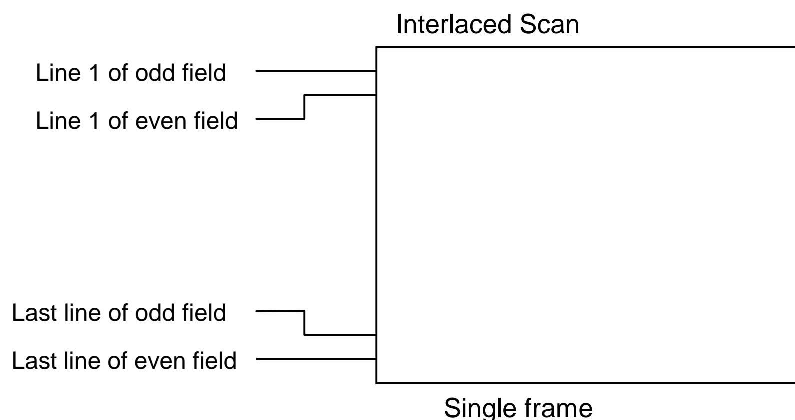
The currently selected analog signal is mapped directly to the BT878 analog video input. This feature is primarily meant for debugging. The video input source must be an analog input (either COMPOSITE or SVIDEO). This feature causes the Osprey-500 to bypass the hardware de-interlacing filter, and therefore the hardware filter option has no effect when Analog Bypass is enabled.

Filtering

The input signal is de-interlaced and motion blurred. Filtering will only be performed on images > 240 lines NTSC (284 lines PAL). All releases of the Osprey-500 drivers support filtering in hardware. Software-based filtering is available in release 2.1 of the Osprey-2000 drivers and release 2.2 or later for all other Osprey products. The hardware-based filtering on the Osprey-500 reduces the load on the host processor as compared to the software-based filtering. Please refer to the Osprey-500 product specification document for more information on the hardware filtering.

Odd/Even Send





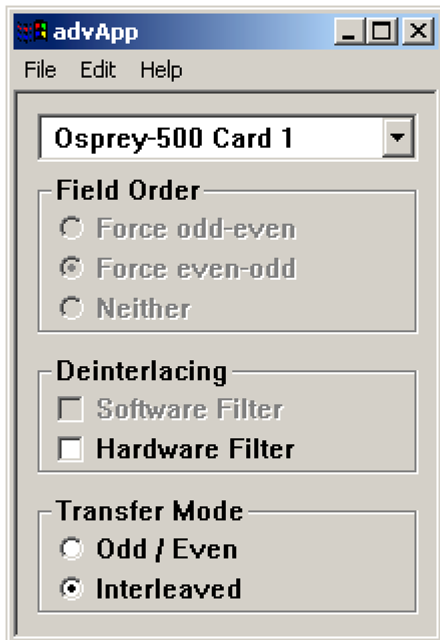
The two diagrams above demonstrate the difference between odd/even scan and interlaced scan. When the image size is > 240 lines (284 lines PAL), setting odd/even scan causes two subsequent fields (the odd followed by the even field) to be placed on top of each other instead of interlaced. For images <= 240 lines, odd/even scan has no effect.

Force Odd/Even Field First

These flags force the order in which the Osprey card combines fields to create a frame. The two flags are mutually exclusive and capture order is indeterminate if both are set. The flags are useful when capturing from progressive scan sources, and the correct flag to use depends on the characteristics of the capture device. If your progressive scan device always sends the odd field of a frame first, then set the **Force Odd Field First** flag. When neither flag is set, the driver will use the first available fields to construct a frame. Setting either flag may increase the time needed to capture a single frame. The increase in capture time is no more than the time required to capture a single field (1/59.94 sec. NTSC or 1/50 sec. PAL).

Release 2.0 of the Osprey-100/200 drivers does not support setting either flag through the SDK. Release 2.1 of the Osprey-500 drivers only supports **Force Odd Field First**. Release 2.1 of the Osprey-2000 drivers and release 2.2 of all other Osprey products support both flags.

Sample Application: advApp



The advApp Application demonstrates using the MMAC_ADV_CONTROL message to get or set some of the Advanced Video Processing features. It allows you to select any Osprey Card in your system and displays the current values of the Field Order, Deinterlacing Filter, and Transfer mode settings. It will also allow you to changes the current settings. Note that not all settings are valid for all Osprey Card Types. Any value that is not valid for the selected card will be grayed out in the advApp GUI.

The Advanced settings should be set to the desired values before starting an application that will use them. For example, if you want to capture video using the vidcap application with the Hardware Filter on, you should click the Hardware Filter box on the advApp application before starting up the vidcap

application. Otherwise the Hardware Filter Setting will not take effect unless you shut down and restart vidcap. Likewise, if you change the advanced settings in another application you will not see the change in advApp unless you shut advApp down and restart it.

Functions

The SDK uses the multimedia functions OpenDriver(), CloseDriver(), and SendDriverMessage() to communicate with the driver. For complete descriptions of these functions, refer to the Visual C++ documentation under *Platform SDK – Graphics and Multimedia Services – Multimedia Reference*. Described below are the specialized parameters and messages used with these functions:

OpenDriver()

```
#include <mmsystem.h>

HDRVR OpenDriver(
  LPCWSTR lpDriverName,
  LPCWSTR lpSectionName,
  LONG    lParam
);
```

Sample usage:

```
#include <o100ext.h>

O100_OPEN oOpen;
HDRVR     hDrvr;

oOpen.dwSizeof = sizeof(oOpen);
oOpen.dwSig    = MMAC_INPUT_SIG;
oOpen.dwFlags  = VID_CHANNEL;
oOpen.dwDevice = 0;

hDrvr = OpenDriver(L"msvideo", NULL, (LPARAM)&oOpen);

if (hDrvr == NULL)
  return FALSE;

if ((oOpen.dwCaps & o100_CAPS_V134) < o100_CAPS_V134)
  return FALSE;
```

Parameters:

The first two parameters are as described in the Microsoft documentation. When the Osprey-100 driver is installed normally, the first and second parameters are set to L"msvideo" and NULL respectively. These parameters mean that the video capture driver is to be identified by the msvideo registry variable, which for the Osprey-100 is set as follows:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ Drivers32\msvideo = o100vc.dll

The third parameter is a pointer to the SDK's O100_OPEN structure. The declaration of O100_OPEN is as follows:

```
typedef struct {
  DWORD dwSizeof;
  DWORD dwSig;
  DWORD dwDevice;
  DWORD dwFlags;
  DWORD dwCaps;
  DWORD dwError;
}
O100_OPEN, *PO100_OPEN, **PPO100_OPEN;
```

On input it is mandatory to set the first four members:

- `dwSizeof` is always the size of the `O100_OPEN` structure.
- `dwSig` is always `MMAC_INPUT_SIG`, defined in <mmacdrv.h>.
- `dwDevice` is the zero-based number of the board that you want to connect to.

- dwFlags is set to VID_CHANNEL, except as specifically noted.

Set dwError to 0 on input if you are going to check the return value.

Return Values:

If OpenDriver() succeeds it returns a handle to the driver. Members of O100_OPEN are set as follows:

- dwSig is set to MMAC_OUTPUT_SIG. You can check this value if you need to be extremely certain that you have accessed the Osprey-100 driver and not some other.
- dwCaps is set to a mask showing what messages the installed version of the driver supports. The numbering of the bits of dwCaps correspond to offset from the base value of the message IDs in <mmacdrv.h>. For example, MMAC_o100_REGISTER_GET has an offset of 10. If bit 10 of dwCaps is set, then the driver supports MMAC_o100_REGISTER_GET. Since driver versions 1.33 and earlier do not support MMAC_o100_REGISTER_GET, they would return bit 10 of dwCaps cleared.

If OpenDriver() fails it returns NULL. If the call fails because the call never reached the Osprey-100 driver, no members of O100_OPEN are changed. If the call reaches the capture driver but the capture driver fails it, the capture driver will set dwError set to one of the following diagnostic codes, as defined in <msvideo.h>:

- DV_ERR_FLAGS: dwFlags is set to an illegal value.
- DV_ERR_ALLOCATED: no more handles to the driver are available.

If the call fails, the contents of the O100_OPEN structure other than dwError should be regarded as indeterminate.

CloseDriver()

```
#include <mmsystem.h>

LRESULT CloseDriver(
  HDRVR hDrvr,
  LONG  lParam1,
  LONG  lParam2
);
```

Sample Usage:

```
CloseDriver(hDrvr, 0, 0);
CloseDriver(hDrvr, 0, CTL_SAVE);
```

Parameters:

`hDrvr` is the handle returned by `OpenDriver()`.

When the third parameter is `CTL_SAVE` instead of `0`, it causes the driver to write to the registry the current video source input values - brightness, contrast, saturation, hue – for the currently selected input.

Return Value:

Nonzero if successful, zero if unsuccessful.

SendDriverMessage()

```
LRESULT SendDriverMessage(
  HDRVR hDrvr,
  UINT  uMsg,
  LONG  lParam1,
  LONG  lParam2
);
```

Sample Usage:

```
// Find out how many boards are installed:
bOk = SendDriverMessage(
  hDrvr,
  MMAC_NDEVICES_QUERY,
  &lNDevices,
  sizeof(LONG)
);

if (!bOk)
  return FALSE;
```

Parameters:

The first parameter is always the handle returned by `OpenDriver()`.

The second parameter is always one of the Osprey-100 messages defined in <mmacdrv.h>.

The third and fourth parameters are message dependent. Their usage with each message is described below.

Return Value:

Nonzero if successful, zero if unsuccessful.

MESSAGES

The following six messages are current and recommended for use. Some of them require Osprey-100 driver versions 1.35 or later, as noted in the individual descriptions.

```
MMAC_NDEVICES_QUERY
MMAC_CAPS_QUERY
MMAC_DEVICEN_SET
MMAC_o100_SRC_CONTROL
MMAC_o100_CC_CONTROL
MMAC_o100_CC_FLAGS_GET
MMAC_o100_CC_CONTROL2
MMAC_o100_VBI_CONFIG
MMAC_o100_VBI_CONTROL
MMAC_o100_VBI_ADD_BUFFER
MMAC_o100_REGISTER_GET
MMAC_o100_REGISTER_SET
MMAC_ADV_CONTROL
```

The following five messages are largely obsolete. There may be occasional instances where they are useful, and they may be necessary to support pre-1.35 versions of the Osprey-100 driver. Future versions of the driver will continue to support them, but we recommend converting to newer messages in most cases.

```
MMAC_o100_SETTINGS_GET
MMAC_o100_SETTINGS_SET
MMAC_o100_SETTINGS_GET2
MMAC_o100_SETTINGS_SET2
```

MMAC_NDEVICES_QUERY

Purpose:

To determine the number of boards installed in the system.

Versions:

Supported by driver versions 1.21 and later.

Sample Usage:

```
// Find out how many boards are installed:
BOOL bOk;
LONG lNDevices;

bOk = SendDriverMessage(
  hDrvr,
  MMAC_NDEVICES_QUERY,
  &lNDevices,
  sizeof(LONG)
);

if (!bOk)
  return FALSE;
```

lParam1:

A pointer to a `LONG`. If the function succeeds, the number of Osprey-100 boards installed in the system is placed in the receiving variable.

lParam2:

Set lParam2 to `sizeof(LONG)`.

MMAC_CAPS_QUERY

Purpose:

1. Retrieve a capabilities mask indicating what commands the currently loaded o100vc.dll supports.
2. Enables further SDK messages to the driver.

Versions:

Supported by driver versions 1.21 and later. Use this message only if your application has to work with driver versions prior to 1.35.

If you do not have this constraint, a cleaner and simpler alternative is to call `OpenDriver()` with a pointer to an `O100_OPEN` structure as its third parameter. The capabilities mask will be returned in the `dwCaps` member of `PO100_OPEN`. Passing an `O100_OPEN` structure to `OpenDriver()` also enables further SDK messages to the driver.

Sample Usage:

```
CAPS_QUERY oCapsQuery;
BOOL bOk;

oCapsQuery.dwSig = MMAC_INPUT_SIG;

bOk = SendDriverMessage(
  hDrvr,
  MMAC_CAPS_QUERY,
  (LONG)&oCapsQuery,
  sizeof(CAPS_QUERY)
);

if (!bOk)
  return FALSE;

// Make sure all commands in the v1.34 driver are supported:
if (
  (oCapsQuery.dwSig != MMAC_OUTPUT_SIG) ||
  ((oCapsQuery.dwCapsMask & o100_CAPS_V134) != o100_CAPS_V134)
)
  return FALSE;
```

lParam1:

A pointer to a `CAPS_QUERY` structure, declared as follows in <mmacdrv.h>:

```
typedef struct {
  DWORD dwSig;
  DWORD dwCapsMask;
}
CAPS_QUERY, *PCAPS_QUERY, **PPCAPS_QUERY;
```

Usage is as follows:

In: `dwSig` = the constant `MMAC_INPUT_SIG`

`dwCapsMask` = unused

Out: `dwSig` = the constant `MMAC_OUTPUT_SIG`

`dwCapsMask` = a mask of the available SDK messages

If bit n of `dwCapsMask` is set, then the message numbered n relative to the SDK message base is supported by the currently loaded driver. The driver will return FALSE from any `SendDriverMessage()` call with a message argument that it does not support.

lParam2:

Set lParam2 to sizeof(CAPS_QUERY).

Description:

This message was designed to ensure that your application will only pass SDK messages to an Osprey-100 driver, not to a different driver that cannot process them. It prevents an application from inadvertently passing messages to the Osprey-100 driver that are not in SDK format. It also returns a mask indicating the available SDK messages.

MMAC_DEVICEN_SET

Purpose:

Set the number of the board to be accessed in the next `capDriverConnect()` call.

Versions:

Supported by driver versions 1.21 and later.

Sample Usage:

See the sample application App2X for a complete usage example.

```
BOOL  bOk;
BOOL  bConnected;
DWORD dwDevice = 0;  // connect to board 0
HWND  hCapWnd;

bOk = SendDriverMessage(hDrvr, MMAC_DEVICEN_SET, dwDevice, 0);
If (!bOk)
  return FALSE;

bConnected = capDriverConnect(hCapWnd, 0);
```

lParam1:

The zero-based number of the board to be opened next.

lParam2:

Not used – set to 0.

Description:

This message is useful in applications that call `capDriverConnect()` multiple times to connect to multiple boards simultaneously.

This message does not change the device that `hDrvr` is associated with, even after `capDriverConnect()` is called to connect VfW to a different board. To change this association, close `hDrvr` and reopen it with a new `o100_OPEN.dwDevice` setting.

Whenever you call `OpenDriver()` with a pointer to an `o100_OPEN` as its third parameter, the next `capDriverconnect()` will connect VfW to `o100_OPEN.dwDevice`. It is unnecessary to send a `MMAC_DEVICEN_SET` message unless you want to open a device different from the one that was set by `OpenDriver()`.

MMAC_o100_SRC_CONTROL

Purpose:

To enumerate and control the video input sources.

- determine the number of Composite, S-Video, SDI or DV inputs.
- select which video input to make active.
- determine and control video format – ntsc, pal, etc.
- determine and control brightness, contrast, hue, and saturation settings for the video source.

Osprey cards come in a number of physical variants – with one or more composite inputs; with or without an S-Video, SDI or DV input, with or without ntsc capability, and with or without pal capability. Physical inputs as laid out on the mounting bracket may correspond to input multiplexer positions in different ways on different board types. `MMAC_o100_SRC_CONTROL` enumerates the capabilities of the hardware, and maps video inputs by logical function rather than by the physical multiplexer settings used in the Bt848 `IFORM` register.

Versions:

Supported by driver versions 1.35 and later.

Sample Usage:

`MMAC_o100_SRC_CONTROL` has numerous subcommands and capabilities that cannot really be indicated by one or two simple examples. Refer to the sample application CtlApp for a wide range of useful examples.

```
// Get information on svideo inputs:
SRCCTL oSrcCtl;
BOOL   bOk;

oSrcCtl.dwCmd      = SVIDEO;
oSrcCtl.dwData     = 0;
oSrcCtl.dwVFmt     = 0;
oSrcCtl.pCtlsGet   = NULL;
oSrcCtl.pCtlsSet   = NULL;

bOk = SendDriverMessage(
  hDrvr,
  MMAC_o100_SRC_CONTROL,
  (LONG)&oSrcCtl,
  sizeof(SRCCTL)
);

if (!bOk)
  return FALSE;
```

lParam1:

A pointer to a `SRCCTL` structure. `SRCCTL` is declared as follows:

```
typedef struct {
  DWORD     dwCmd;      /* input command, output data           */
  DWORD     dwData;     /* input and output data parameter      */
```

```
  DWORD     dwVFmt;     /* video format to get or set           */
  PCONTROLS pCtlsGet;   /* brightness, etc, to get from driver  */
  PCONTROLS pCtlsSet;   /* brightness, etc, to set in   driver  */
}
SRCCTL, *PSRCCTL, **PPSRCCTL;
```

The CONTROLS structure referenced in SRCCTL is declared as follows:

```
typedef struct {
  WORD wContrast;       /* range 0..511 (0..0x1FF), default 216 (0xD8) */
  WORD wSatU;           /* range 0..511 (0..0x1FF), default 254 (0xFE) */
  WORD wSatV;           /* range 0..511 (0..0x1FF), default 180 (0xB4) */
  CHAR cBrightness;     /* range -128..+127 (0x80..0x7F), default 0    */
  CHAR cHue;            /* range -128..+127 (0x80..0x7F), default 0    */
}
CONTROLS, *PCONTROLS, **PPCONTROLS;
```

Note that the normally wSatU and wSatV exist in the following fixed relationship:

```
oCtls.wSatV = oCtls.wSatU * 0xB4 / 0xFE;
```

lParam2:

Set `lParam2` to `sizeof(SRCCTL)`.

Command Formats:

This message controls three groups of parameters

a. Enumeration and selection of the logical video input.

b. Enumeration and Selection of the video format (ntsc / pal, etc.).

c. Reading and control of the video settings – brightness, contrast, saturation, hue.

A single message can operate on one, two, or all three of these groups. Some combinations are not possible – in particular, you cannot enumerate video inputs and video formats in the same command. But once enumeration is completed, you could, for example, change to a new video input, change to the different video format associated with that input, and set the brightness and hue desired for that input, all in a single message.

The following are the command constants for `dwCmd`:

```
#define COMPOSITE           1
#define SVIDEO              2
#define VIDEOFMT            3
#define VIDEOCTLS           4
#define VIDEO_SOURCE_SDI    0x10
#define VIDEO_SOURCE_DV     0x11
#define VIDEO_SOURCE_MPEG   0x12
```

The Osprey-500 and Osprey-2000 support `VIDEO_SOURCE_SDI` and `VIDEO_SOURCE_DV`. The Osprey-2000 supports `VIDEO_SOURCE_MPEG`. On the Osprey-2000 `VIDEO_SOURCE_MPEG` refers to the uncompressed video generated by the MPEG decoder.

The six primary command formats are as follows:

1. *To get information about the video input multiplexer logical settings:*

in: dwCmd = COMPOSITE, SVIDEO, VIDEO_SOURCE_SDI, VIDEO_SOURCE_DV or VIDEO_SOURCE_MPEG
dwData = 0
dwVFmt = see below**
pCtlsGet = see below***
pCtlsSet = see below****
out: dwCmd = number of video inputs
dwData = 1..4 if input of dwCmd's type is now selected
= 0 if an input of another type is now selected
dwVFmt = see below**
pCtlsGet = see below***
pCtlsSet = see below****

2. *To set the video input multiplexer:*

in: dwCmd = COMPOSITE, SVIDEO, VIDEO_SOURCE_SDI, VIDEO_SOURCE_DV or VIDEO_SOURCE_MPEG
dwData = which logical input of dwCmd's type to switch to, 1..4
dwVFmt = see below**
pCtlsGet = see below***
pCtlsSet = see below****

out: dwCmd = unchanged
dwData = which logical input of dwCmd's type is now selected, 1..4
= 0 is an error - you tried to access a non-existent input
dwVFmt = see below**
pCtlsGet = see below***
pCtlsSet = see below****

3. *To get video input NTSC/PAL format information:*

in: dwCmd = VIDEOFMT command constant
dwData = unused
dwVFmt = 0
pCtlsGet = see below***
pCtlsSet = see below****

out: dwCmd = unchanged
dwData = bits 1..7 set for supported formats 1..7 (see below)
dwVFmt = see below**
pCtlsGet = see below***
pCtlsSet = see below****

Note that you can get or set the video format for the currently selected input with any of the other commands. Use the VIDEOFMT command when either

a. You want to retrieve the mask of supported formats in dwData.

b. You want to get or set the video format without taking any other action such as changing the video input.

4. *To set video input NTSC/PAL format:*

in: dwCmd = VIDEOFMT command constant
dwData = unused
dwVFmt = 1..7 see below**
pCtlsGet = see below***
pCtlsSet = see below****

out: `dwCmd` = unchanged
`dwData` = unchanged
`dwVFmt` = see below**
`pCtlsGet` = see below***
`pCtlsSet` = see below****

5. *To get/set controls - brightness, contrast, hue, saturation:*

in: `dwCmd` = `VIDEOCTLS` command constant
`dwData` = see below*****
`dwVFmt` = unused
`pCtlsGet` = see below***
`pCtlsSet` = see below****

out: `dwCmd` = unchanged
`dwData` = see below*****
`dwVFmt` = unused
`pCtlsGet` = see below***
`pCtlsSet` = see below****

6. *To get/set video pixel size:*

in: `dwCmd` = `VIDEOPIXELRATIO` command constant
dwData = unused
dwVFmt = 0, SRC_SQUARE or SRC_CCIR601 see below **
pCtlsGet = see below***
pCtlsSet = see below****

out: dwCmd = unchanged
dwData = unchanged
dwVFmt = see below**
pCtlsGet = see below***
pCtlsSet = see below****

Note that you can get or set video controls with any `dwCmd` setting. To do so, you point `pCtlsGet` and/or `pCtlsSet` to a `CONTROLS` structure. The purpose of the `VIDEOCTLS` command constant is to allow you to get or set video controls without doing anything else in the command or to bypass color correction.

** `dwVFmt` 1..7 are the seven video input formats supported by the Bt848 / Bt878:

```
#define SRC_NTSC      1
#define SRC_NTSCJ     2
#define SRC_PAL       3
#define SRC_PALM      4
#define SRC_PALN      5
#define SRC_SECAM     6
#define SRC_PALNC     7
```

`dwVFmt` 0 on input == don't change the format
`dwVFmt` 1..7 on input == change the format to 1..7
`dwVFmt` 1..7 on output == the video format currently selected

For the VIDEOPIXELRATIO command only:

dwVFmt 0x10,0x20	on input	== change the aspect ratio
dwVFmt 0	on input	== don't change the aspect ratio
dwVFmt	on output	== the aspect ratio currently selected

*** if pCtlsGet is non-NULL, current control settings are returned. If the message is changing the video source multiplexer, the control settings returned will be those associated with the new multiplexer setting.

**** if pCtlsSet is non-NULL, current control settings are set to the contents of pCtlSet->. If the message is also changing the video source multiplexer, the control settings associated with the new rather than the old multiplexer setting will be changed.

*****For VIDEOCTLS command, this field controls the bypass color correction feature available on the Osprey-500 and –2000 class cards. Enabling color correction bypass causes the driver to disable all hardware-based color correction.

dwData on input:

CTLS_QUERY	==	query current setting
CTLS_BYPASS	==	bypass control settings
CTLS_USE	==	use control settings

The return value of dwData always contains the current bypass setting (either CTLS_BYPASS or CTLS_USE).

MMAC_o100_CC_CONTROL

Purpose:

To enable and disable Closed Captioning in the driver, to set CC options, and to control callbacks with CC text to the application.

Versions:

Supported by driver versions 1.21 and later. Significant corrections are included in versions 1.40 and later.

Sample Usage:

Refer to the sample application CCApp for a complete example of how to use this message.

```
void CALLBACK CCLines(PVOID pv, DWORD dw);

UCHAR           aucBuf[33];
CC_CALLBACK_CTL oCC;
BOOL            bOk;

oCC.dwFlags    = CB_ENABLE | CB_DLG_OVERRD | CB_DLG_DSBL;
oCC.pFun       = CCLines;
oCC.pvContext  = NULL;
oCC.pbBuf      = aucBuf;

bOk = SendDriverMessage(
  hDrvr,
  MMAC_o100_CC_CONTROL,
  (LPARAM)&oCC,
  sizeof(CC_CALLBACK_CTL)
);

if (!bOk)
  return FALSE;
```

lParam1:

Points to an initialized `CC_CALLBACK_CTL` structure, declared as follows in <o100ext.h>:

```
typedef struct {
  DWORD dwFlags;      /* flags                        */
  VOID  (CALLBACK *pFun)(PVOID, DWORD);
                      /* pointer to callback function */
  PVOID pvContext;    /* pointer to app's context     */
  PBYTE pbBuf;        /* pointer to app's line buffer */
}
CC_CALLBACK_CTL, *PCC_CALLBACK_CTL;
```

The bits of `dwFlags` are defined as follows:

```
#define CB_ENABLE       0x01
#define CB_RAW          0x02
#define CB_DLG_OVERRD   0x04
#define CB_DLG_DSBL     0x08
```

```
#define CB_RQ_CHAN2      0x10
#define CB_RQ_FIELD2     0x20
#define CB_RQ_TXTMODE    0x40
#define CB_DSPL_INHIBIT  0x80
```

lParam2:

Set `lParam2` to `sizeof(CC_CALLBACK_CTL)`.

Description:

This SDK does not attempt to document the details of the Closed Captioning standard. If you are accessing CC text in fully decoded line form, you will not need further documentation. If you are working with raw character pair data, you will need to refer to documentation from the Closed Captioning Institute or elsewhere.

The control bits in the `dwFlags` member of `CC_CALLBACK_CTL` have meanings as follows:

- `CB_ENABLE`: When set, CC is enabled and callbacks to your application are enabled; your application receives CC text.

  Note - you will see CC text on video only if either (1) `CB_DLG_OVERRD` is set and `CB_DSPL_INHIBIT` is cleared, or (2), `CB_DLG_OVERRD` is cleared and the driver's CC dialog "Display on Screen – Enable" control is checked.

  When `CB_ENABLE` is cleared, CC callbacks to your application are disabled. The seven flags described below have no effect. The driver's CC control dialog settings control the display of CC.

  Note – you can control CC without receiving callbacks by setting CB_ENABLE and setting the `pFun` member of `CC_CALLBACK_CTL` to NULL.

- `CB_RAW`: When set, your callbacks will contain raw CC codes. When cleared your callbacks will contain lines of ANSI ASCII text.
- `CB_DLG_OVERRD`: When set, the five flags below override settings the user may have set in the driver's CC ("Display") control dialog. If this flag is not set, the five flags described below have no effect; the settings from the driver's CC control dialog are used instead.

  The application never overrides the driver's "Over Video / Own Window", "Save to File", or "TextColor" settings regardless of `CB_DLG_OVERRD's setting.`

- `CB_DLG_DSBL`: If set, the user cannot override the CC settings you provide. The driver's CC dialog controls are grayed out. If not set, the driver responds both to inputs the driver's dialog and to `CC_CALLBACK_CTL` messages from the application – whichever was most recently changed will take precedence.
- `CB_DSPL_INHIBIT`: If set, CC text in overlay mode will not be displayed on the video window or separate text window, and in capture mode will not be painted on the captured video frames. If this flag is clear, overlay and capture will display CC text whenever `CB_ENABLE` is set. This bit is the inverse of the driver's CC dialog checkbox, Display on Screen – Enable.
- `CB_RQ_CHAN2`: When set selects the alternate channel.
- `CB_RQ_FIELD2`: When set selects the alternate odd/even field.
- `CB_RQ_TXTMODE`: When set selects text mode CC rather than standard video CC.

The last three flags are cleared for most CC content. The options they control are described in more detail in CC standards documents.

`pFun` is a pointer to your callback function. How this callback works is described below. If you want to control CC from your application using the `dwFlags` bits but do not want to receive callbacks, set `pFun` to NULL.

`pvContext` is a context pointer you can pass to the driver when you set up the callback. It is passed back to your app in both the raw and line mode callbacks. If you do not need a context pointer you can set `pvContext` to NULL.

`pbBuf` is a pointer to a character array. In line mode, `pbBuf` must point to a space allocated by your app that is at least 33 bytes long that will receive null-terminated lines of CC text. In raw mode, `pbBuf` can be NULL.

The Osprey-100 driver calls `pFun` to pass CC data to you. If the `CB_RAW` bit of `dwFlags` is set, each callback passes a single character pair to you. If the `CB_RAW` flag is cleared, each callback passes a line of decoded CC text to you. In both cases the prototype of your function is the same:

```
void CALLBACK CCCallback(PVOID pv, DWORD dw);
```

The callback's parameters are interpreted as follows:

`CB_RAW` set:

`PVOID pv`:	the context pointer you passed to the driver in `CC_CALLBACK_CTL`
`DWORD dw`:	bits 0..7 contain the first character of the pair bits 8..15 contain the second character

`CB_RAW` cleared:

`PVOID pv`:	the context pointer you passed to the driver in `CC_CALLBACK_CTL`
`DWORD dw`:	not used

Here are minimalist examples of CC callback functions:

```
void CALLBACK CCRaw(PVOID pv, DWORD dw)
{
  printf("%c%c\n", dw & 0xFF, dw >> 8);
}

void CALLBACK CCLines(PVOID pv, DWORD dw)
{
  printf("%s\n", aucBuf);
}
```

In `CCLines()`, `aucBuf[]` is the buffer that was originally passed to the driver in `CC_CALLBACK_CTL`.

MMAC_o100_CC_FLAGS_GET

Purpose:

To read the current settings of the video capture driver's CC flags bits.

Versions:

Supported by driver versions 1.40 and later.

Sample Usage:

```
/* Taken from CCApp, CdDlg.cpp, CCcDlg::CCRead(). */
/* Works with driver versions >= 1.40 only. */
/* If CB_DLG_OVERRD is set, ignore the driver's bits: */
if (
  hDrvr &&
  ((oOpen.dwCaps & o100_CAPS_V140) >= o100_CAPS_V140) &&
  !(oCC.dwFlags & CB_DLG_OVERRD)
){
  DWORD dwReadFlags;
  BOOL  bOk = SendDriverMessage(
    hDrvr,
    MMAC_o100_CC_FLAGS_GET,
    (LPARAM)&dwReadFlags,
    sizeof(DWORD)
  );

  /* Merge the app-controlled bits with the jointly-controlled bits: */
  if (bOk) {
    oCC.dwFlags =
      (oCC.dwFlags & CB_APP_BITS    ) |
      (dwReadFlags & CB_DRIVER_BITS);
  }
}
```

lParam1:

Points to a `DWORD` that will receive the flag bits. The receiving variable is set only if the call succeeds, indicated by a return value of 1. Note that, as explained below, only four of the eight CC flag bits are assigned; the rest will be zero.

lParam2:

Set `lParam2` to `sizeof(DWORD)`.

Description:

There are two groups of CC flag bits: Those that are controlled by the application only, not by the video capture driver; and those that may be controlled by the video capture driver if the `CB_DLG_OVERRD` bit is cleared, as well as by the application.

The bits controlled by the application only are as the following:

```
#define CB_ENABLE       0x01
#define CB_RAW          0x02
#define CB_DLG_OVERRD   0x04
#define CB_DLG_DSBL     0x08
```

The bits controllable by both the application and driver are the following:

```
#define CB_RQ_CHAN2     0x10
#define CB_RQ_FIELD2    0x20
#define CB_RQ_TXTMODE   0x40
#define CB_DSPL_INHIBIT 0x80
```

`MMAC_o100_CC_FLAGS_GET` returns only the bits jointly controlled by application and driver; the application-only bits are returned as zero.

MMAC_o100_CC_CONTROL2

Purpose:

Augments the older command `MMAC_o100_CC_CONTROL` to provide improved control of Closed Captioning. `MMAC_o100_CC_CONTROL` continues to be supported in newer drivers.

Includes timestamps and raw mode filtering options.

Allows CC capture without requiring simultaneous video preview, overlay, or capture.

Allows multiple simultaneous CC callbacks:

- odd field / even field
- raw mode / line mode
- text mode / video mode
- channel 1 / channel 2

Allows CC callbacks from one or more processes separate from the main Video for Windows application accessing the card.

Provides a pointer to the current CC screen array for complete positioning and character attribute information.

Versions:

Supported by driver versions 1.50 and later.

lParam1:

Points to an initialized `CC_CALLBACK_CTL2` structure, declared as follows in "o100ext.h":

```
typedef struct {
  DWORD dwCmd;      /* command code - see below      */
  VOID  (CALLBACK *pFun)(PCC_CALLBACK2);
                    /* pointer to callback function  */
  PVOID pContext;   /* pointer to app’s context data */
  PCHAR pcData;     /* pointer to app’s line buffer  */
}
CC_CALLBACK_CTL2, *PCC_CALLBACK_CTL2;
```

The declaration of the `CC_CALLBACK2` structure referenced here is shown later in this section.

lParam2:

Set `lParam2` to `sizeof(CC_CALLBACK_CTL2)`.

Command Formats:

A command is defined by the `dwCmd` parameter. The command bits of `dwCmd` are as follows:

```
#define CC2_CMD_MASK    0x000F0000  /* command nybble                  */
#define CC2_GET_DISPLAY 0x00010000  /* get current CC display settings  */
#define CC2_SET_DISPLAY 0x00020000  /* set new CC display settings      */
#define CC2_RESTORE     0x00030000  /* restore CC display settings      */
#define CC2_RAW_OPEN    0x00040000  /* open  a raw  mode callback       */
#define CC2_RAW_CLOSE   0x00050000  /* close a raw  mode callback       */
```

```
#define CC2_LINE_OPEN   0x00060000  /* open  a line mode callback       */
#define CC2_LINE_CLOSE  0x00070000  /* close a line mode callback       */
```

The following additional modifier bits in `dwCmd` are also defined:

```
#define CC2_VIDEO       0x00000002  /* 1 = CC to driver video requested */
#define CC2_FILE        0x00000004  /* 1 = CC to driver file  requested */
#define CC2_TEXTMODE    0x00000010  /* 1 = CC text mode requested       */
#define CC2_CHAN2       0x00000020  /* 1 = CC channel 2 requested       */
#define CC2_FIELD2      0x00000040  /* 1 = CC field 2 requested         */
#define CC2_TEXTWND     0x00000080  /* 1 = CC to driver text window rq  */
#define CC2_UNFILTERED  0x00000200  /* 1 = CC return raw null pairs     */
```

In the following format descriptions, the bits of `dwCmd` are represented in binary format – for example,

`0010 xxxx xxxx WFCT xPVx`.

Each bit may be shown as 0, 1, ‘x’, or an uppercase letter.

- ***0 and 1*** define the overall command - `CC2_GET_DISPLAY`, etc.
- ***‘x’*** is a don’t-care or not-used bit, but on input should be always set to zero to ensure future compatibility.

The letters `VPTCFWU` have meanings as follows:

- ***V – CC2_VIDEO - display CC***. If 1, the driver will display CC on the screen.
- ***P – CC2_FILE - print to file.*** If 1, the driver will print CC lines to the user-selectable file.
- ***T – CC2_TEXTMODE - text/video.*** If 0, normal video-mode CC will be returned. If 1, special text-mode CC will be returned.
- ***C – CC2_CHAN2 - channel.*** If 0, CC channel 1 data will be returned. If 1, CC channel 2 data will be returned.
- ***F – CC2_FIELD2 - even/odd field.*** Determines whether CC for this channel will come from even or odd fields. 1 = even, 0 = odd.
- ***W – CC2_TEXTWND - own window/on video.*** If 0, the driver will display CC on the video in the normal way. If 1, the driver will display CC in a separate text window. Not meaningful if D (display CC) is 0.
- ***U – CC2_UNFILTERED - filter bit for raw mode callbacks.*** 1 = return all character pairs, including null pairs. 0 = don’t return null pairs.

There are seven primary commands:

Commands 1, 2, and 3 deal with the capture driver’s “native” CC settings – that is, the settings that are controlled from the user’s CC dialog box and which control CC display on the screen. They allow you to override the user’s settings and to restore them.

Commands 1, 2, and 3 can only be used from the primary process, usually one that opens the driver in Video for Windows mode.

Commands 4, 5, 6, and 7 allow you to control callback channels to your application. You can have multiple callback channels with settings distinct from the driver’s “native” CC settings.

Commands 4 - 7 can be used from the primary a process, or from a secondary process that does not open the driver in Video for Windows mode. To use commands 4 - 7 from a secondary process, open the driver with the CC2_CHANNEL flag set rather than the VID_CHANNEL flag.

Command 1. Query the driver’s current CC settings.

In: `dwCmd` = `0001 xxxx xxxx xxxx xxxx`
`pFun` = NULL

`pContext` = NULL
`pcData` = NULL

Out: `dwCmd` = `0001 xxxx xxxx WFCT xPVx` – the driver's current CC settings
`pFun` = unchanged
`pContext` = unchanged
`pcData` = unchanged

Command 2. Set driver's CC settings.

In: `dwCmd` = `0010 xxxx xxxx WFCT xPVx` – the new settings
`pFun` = NULL
`pContext` = NULL
`pcData` = NULL

Out: `dwCmd` = unchanged
`pFun` = unchanged
`pContext` = unchanged
`pcData` = unchanged

Command 2 overrides the driver's CC settings from the registry with the settings in `dwCmd`, and disables the driver's CC dialog box.

Command 3. Restore CC settings back to the driver's internal settings.

In: `dwCmd` = `0011 xxxx xxxx xxxx xxxx`
`pFun` = NULL
`pContext` = NULL
`pcData` = NULL

Out: `dwCmd` = `0011 xxxx xxxx WFCT xPVx` – the restored settings
`pFun` = unchanged
`pContext` = unchanged
`pcData` = unchanged

Command 3 restores the driver's CC settings from the registry and re-enables its CC dialog box.

Commands 4 – 7 all operate on a specific CC "streams". A CC "stream" is a combination of even/odd, raw mode/line mode, video/text, and channel1/channel2. There are 10 possible stream types – two for raw mode capture, eight for line mode. Each process can open up to one of each stream type. The `dwCmd` parameter specifies which of the ten streams you are operating on, and whether you are turning it on or off.

For raw-mode capture, there are two possible streams – field 1 or field 2. There are two options – unfiltered and filtered. Each raw-mode callback returns one raw character pair, which may be an ASCII text pair, a CC command code, or a high-level diagnostic code. The details of this mode are given in a separate section below.

Command 4. Open raw mode stream.

In: `dwCmd` = `0100 xxxx xxxx xxUx xFxx`
`pFun` = pointer to callback function
`pContext` = pointer to callback context data, or NULL
`pcData` = NULL

Out: `dwCmd` = unchanged
`pFun` = unchanged
`pContext` = unchanged
`pcData` = unchanged

Command 5. Close raw mode stream.

In: `dwCmd` = `0101 xxxx xxxx xxxx xFxx`
`pFun` = not used
`pContext` = not used
`pcData` = not used

Out: `dwCmd` = unchanged
`pFun` = unchanged
`pContext` = unchanged
`pcData` = unchanged

For line-mode capture, there are eight possible streams formed of the possible combinations of (field 1 | field 2) * (video | text) * (channel 1 | channel 2). The returned data is filtered for the combination specified. The details of line mode are described in a section below.

Command 6. Open line mode stream.

In: `dwCmd` = `0110 xxxx xxxx xxxx xFCT`
`pFun` = pointer to callback function
`pContext` = pointer to callback context data, or NULL
`pcData` = pointer to 33-byte character line buffer, or NULL

Out: `dwCmd` = unchanged
`pFun` = unchanged
`pContext` = unchanged
`pcData` = unchanged

Command 7. Close line mode stream.

In: `dwCmd` = `0111 xxxx xxxx xxxx xFCT`
`pFun` = not used – set to NULL
`pContext` = not used – set to NULL
`pcData` = not used – set to NULL

Out: `dwCmd` = unchanged
`pFun` = unchanged
`pContext` = unchanged
`pcData` = unchanged

Callbacks:

CC2 callback functions have a single parameter and are declared as follows:

```
VOID CALLBACK CCCallback(PCC_CALLBACK2);
```

The CC_CALLBACK2 structure is declared as follows:

```
typedef struct {
  DWORD dwFlags;    /* channel flags - see below                      */
  PVOID pContext;   /* optional pointer to app's context, or NULL     */
```

```
    DWORD dwMsec;      /* time of capture                                    */
    PWORD pwScreen;    /* line mode pointer to screen data                   */
    DWORD dwData;      /* line mode CC mode, raw mode character pair         */
  }
  CC_CALLBACK2, *PCC_CALLBACK2;
```

Usage of `CC_CALLBACK2` varies according to whether it is used for raw-mode or line-mode capture:

For raw-mode capture:

- `dwCmd`: the only meaningful flag bit is `CC2_FIELD2`.
- `dwMsec` is the capture time of the character pair, in milliseconds.
- `pwScreen` is unused.
- `dwData` is the character pair. Bits 0 - 7 contain the first character of the pair; bits 8 - 15 contain the second character. See the description above for additional details about the character pair format.

For line-mode capture:

- The CC data line is written to the `pcData` buffer provided in Command 6. See the additional description below.
- `dcCmd`: bits `CC2_FIELD2`, `CC2_CHAN2`, and `CC2_TEXT` of `dwFlags` are meaningful.
- `dwMsec` is the capture time in milliseconds. See the detailed explanation below.
- `pwScreen` is a pointer to the current screen data – see below.
- `dwData` describes the current CC mode – PAINTON, ROLLUP, or POPON – and the type of callback. See below.

You can specify a single `pFun` callback pointer for multiple CC streams when opening them. In this mode of operation the `dwFlags` bits can be used to track what type of data that is being returned. The `pcData` pointer should point to a different buffer for each of multiple streams, however.

Timestamps

Timestamps are in milliseconds relative to when the first CC pair was captured in the current session of your process. The time-zero event could be caused by a `MMAC_o100_CC_CONTROL2` message sent to the driver by your app, or it could be because the driver's native CC interpreter was opened due to pre-existing dialog settings. In other words, you cannot assume that the first timestamp received on a given channel will be 0.

If you close all CC channels to your app, including the driver's internal interpreter, and reopen them, time zero will be reset to the time of capture of the first CC pair by any channel after reopening.

Timestamps are implemented this way so all CC channels of your app will have the same relative timing. You can therefore correlate the timestamps of different channels if need be. If you want different relative time offsets for different channels, you can implement them in your application.

If multiple processes are accessing CC from the same device, time zero will be different for each process.

The exact interpretation of the `dwMsec` timestamp returned in a callback varies according to the format and mode in effect at the time of the callback. These details are given below for each format.

Raw Mode Format

Each raw-mode callback returns one raw character pair, which may be an ASCII text pair, a CC command code, or a high-level diagnostic code. The high-level formats for returned pairs are as follows:

- `0x0000` – CC carrier present, null code.
- `0x8080` - no CC carrier, or completely indecipherable CC data.

- Bits 7 and 15 both high - parity errors in both characters of pair.
- Bit 7 low, bit 15 high - first character valid, parity error in second character.
- Bit 7 high, bit 15 low - parity error in first character, second character valid.
- Bits 7 and 15 both low - both characters valid.

In unfiltered mode, all of these types are returned – there is always a callback for every field 1 or every field 2. Specify unfiltered mode by setting the `CC2_UNFILTERED` bit of `dwCmd`.

In filtered mode, only the last three of the above types are returned – that is, character pairs in which one or both characters are valid. Specify unfiltered mode by clearing the `CC2_UNFILTERED` bit of `dwCmd`.

When a parity error occurs, the seven lower bits of the character are the decoder's best guess as to the intended data.

Line Mode Format

The line mode format is somewhat different from that used in the original `MMAC_o100_CC_CONTROL` protocol.

The behavior of line mode callbacks varies depending on which CC mode is in effect. There are three common modes – *paint-on, rollup, and pop-on*. The `dwData` member of the callback structure indicates the current CC video mode, as well as the completion status of the line.

```
#define CC2_MODEMASK    0x0000000F  /* CC mode field                */
#define CC2_PAINTON     0x00000004  /* CC paint-on mode in effect      */
#define CC2_ROLLUP2     0x00000009  /* CC rollup 2 mode in effect      */
#define CC2_ROLLUP3     0x0000000A  /* CC rollup 3 mode in effect      */
#define CC2_ROLLUP4     0x0000000B  /* CC rollup 4 mode in effect      */
#define CC2_POPON       0x0000000C  /* CC popon    mode in effect      */
#define CC2_NEWLINE     0x00000010  /* This callback is a complete line */
#define CC2_POPPED      0x00000020  /* This callback completes a popon  */
```

Paint-on, Rollup, and Text Modes

Paint-on CC is a protocol in which characters are added to the screen on a one-by-on basis. Rollup mode is the mode frequently seen in broadcast content such as newscasts, sportscasts, and afternoon dramas. A scrolling region of 2, 3, or 4 lines is defined and the text is scrolled up in that region. Text mode is a rarely used mode in which the entire screen is devoted to scrolling CC text; text mode is accessed by opening a text mode channel rather than a video mode channel.

When a CC Paint-on or Rollup mode is in effect, the callback is made whenever the current line changes – not just when a line is completed. The reason is that the CC protocol in these modes does not give any indication that a line is complete. For example, a CC newline code may not be received until long after the line's text has been received and displayed. Some apps therefore will need to look at the line's text before it is completed by a newline or other CC control code.

Each time a new line is started, positions 0 – 31 of the `pcData` buffer are padded with space characters. Position [32] is set to '\0'.

At each callback, the `pcData` buffer contains one or two new characters added to the buffer. `pcData` continues to be space-padded in positions 0 - 31, and position 32 continues to be set to '\0'.

The entire line is presented each time because the CC protocol allows horizontal positioning commands in the course of building a line that could in theory be complex. Unless you know that your application will only receive a simple and constrained set of codes, do not assume that that each callback will simply append one more character at the end of the line.

When the CC decoder encounters a CC control code that starts a new line, it makes the callback with the bit `CC2_NEWLINE` in `dwData` set. The main CC codes that trigger a newline are *erase display*, *carriage*

return, and any *preamble* that alters the vertical position. In Paint-on and Rollup modes, newlines are generally returned with blank lines of data. (In Pop-on mode, described below, newlines are generally returned with full lines of data.)

Therefore, most applications will need to copy the `pcdata` buffer to a separate private buffer each time the callback is made. They can then process the completed line when a `CC2_NEWLINE` flag is received. Note, however, that at the end of the session, there may be no final newline code in the data stream and therefore no `CC2_NEWLINE` in the final callback. The app should therefore process the interim data in the final callback before shutting down.

Applications should generally not alter the data in `pcData`. The driver adds data incrementally to it, rather than rebuilding it from scratch before each callback.

In Paint-on and Rollup CC modes, `dwMsec` is the timestamp for the event that triggered the current callback. If `CC2_NEWLINE` is not set, the timestamp is usually associated with new character data added to the line. If `CC2_NEWLINE` is set, the timestamp is associated with the CC return code or vertical positioning code that starts the new line. As previously stated, this code may be received much later than the text of the preceding line. The recommended timestamp to use for a line of text is either the first callback after the preceding `CC2_NEWLINE` callback, or the last callback before the succeeding `CC2_NEWLINE` callback.

Pop-on Mode

Pop-on CC is the type of CC used in cinematic content where caption blocks are pre-buffered and then popped onto the screen all at once when a CC *flip memory* code is received.

When CC Pop-on mode is in effect, the decoder buffers all incoming characters until it receives a flip memory command in the CC data stream. At that point it makes callbacks, one per line of the Pop-on. All callback lines are padded with space characters out to their full width of 32 printing characters, and null-terminated. In the callbacks for all lines of the Pop-on, the `CC2_NEWLINE` bit of dwData is set. In the callback for the final line, the `CC2_POPPED` bit is set as well as the `CC2_NEWLINE` bit. Each Pop-on is preceded by one blank line with the `CC2_NEWLINE` bit set.

In Pop-on mode, dwMsec is the time of the flip memory CC code.

Screen Pointer and CC Mode

Line mode callbacks include a pointer to the current screen. The screen data is represented as 16-bit words containing the ASCII codes in the lower byte and attribute data in the upper byte. The screen data array is of size [15][32]. The screen pointer is a 16-bit WORD pointer to the first element of the screen data. The format of the data words is as follows, from <o100ext.h>.

```
#define CC_CHAR      0x00FF  /* ascii character field  */
#define CC_COLOR     0x0700  /* color field            */
#define CC_WHITE     0x0000  /* the colors             */
#define CC_GREEN     0x0100
#define CC_BLUE      0x0200
#define CC_CYAN      0x0300
#define CC_RED       0x0400
#define CC_YELLOW    0x0500
#define CC_MAGENTA   0x0600
#define CC_UNDERLN   0x0800  /* underlined             */
#define CC_ITALICS   0x1000  /* italicized             */
#define CC_ATTR_MARK 0x2000  /* used internally by driver, don't change */
#define CC_FLASH     0x4000  /* flashing character     */
```

The pointer address of the current screen may change from callback to callback – particularly in pop-on mode. The pointer and the screen data are guaranteed not to change in the duration of the callback.

The screen data is officially read only. On an experimental basis, it may be possible to modify the data and have the driver display it. When and whether the driver will display the changes is not guaranteed, however.

If you want to work only with the attributed screen data, not with the ASCII line data, you can specify `pcData` as NULL in the open line mode stream command. Doing so avoids a small amount of driver overhead.

Relation of MMAC_o100_CC_CONTROL2 to MMAC_o100_CC_CONTROL

The old format is still supported in newer drivers that support the new format. The two commands can be used independently and at the same time. For example, you could use the old command to capture the CC being displayed on screen and use the new command to also capture data from the other field or other channel.

MMAC_o100_VBI_CONFIG

Purpose:

Configures the interface to raw Vertical Blanking Interval (VBI) data. See the description of the VbiGraph sample app for a general description of how the VBI interface works.

Versions:

Supported by driver versions 1.50 and later.

lParam1:

Points to an initialized `VBIFMT` structure, declared as follows in "o100ext.h":

```
typedef struct {
  DWORD   dwRev;          /* sw rev that buffer conforms to             */
  DWORD   dwFlags;        /* command and status flags - see below       */
  DWORD   dwDriverID;     /* vfw driver id - not used for sdk interface */
  DWORD   dwFormat;       /* video format - see below                   */
  DWORD   dwStartLine;    /* first VBI line                             */
  DWORD   dwEndLine;      /* last  VBI line, inclusive                  */
  DWORD   dwHDelay;       /* VBI_HDELAY raw Bt848/878 setting           */
  DWORD   dwHDwords;      /* VBI_PKT    raw Bt848/878 setting           */
  DWORD   dwBufBytes;     /* size of VBIHDR plus attached data          */
  VOID    (CALLBACK *pCallback)(PVOID, PVBIHDR);
                          /* where to call back with vbi data           */
  PVOID   pContext;       /* optional pointer to client's context data  */
}
VBIFMT, *PVBIFMT, **PPVBIFMT;
```

The members of `VBIFMT` should be set as follows on input . No values are changed on output.

- `dwRev`: Set to 0.
- `dwFlags`: Set to 0.
- `dwDriverID`: Set to 0. Not used for SDK interface.
- `dwFormat`: Set to zero or to one of the following values:

```
#define VBI_NTSC    0x01      /* NTSC */
#define VBI_NTSCJ   0x02      /* NTSC-J */
#define VBI_PAL     0x03      /* PAL-BDGHI */
#define VBI_PALM    0x04      /* PAL-M */
#define VBI_PALN    0x05      /* PAL-N */
#define VBI_SECAM   0x06      /* SECAM */
#define VBI_PALNC   0x07      /* PAL-Nc */
```

  If `dwFormat` is set to zero the driver's currently selected format will be used. If `dwFormat` is set to one of the above #defines, that mode will be set for the duration of the VBI session, provided that another stream is not already accessing the board using another format. If such a conflict occurs, the `MMAC_o100_VBI_CONFIG` message will fail.

- `dwStartLine`: The zero-based start line of the block of VBI lines to be captured. The numbering is in accordance with the Bt848/878 data sheet.

- `dwEndLine`: The zero-based end line of the block of VBI lines to be captured. `dwEndLine` must be greater than or equal to `dwStartLine`. The VBI data block includes the line numbered `dwEndLine`, so the number of VBI lines is `dwStartLine` + 1 – `dwEndLine`. The current version of the driver does not allow the end line to be greater than the first line of video (It can equal that line.). The line number of the first line of video is 11 for NTSC, and 16 for PAL. Line 11 of NTSC is the Closed Caption line. In the current version of the driver, the NTSC Closed Caption line can only go to one stream – the decoded Closed Caption stream, or the raw VBI stream. Whichever of these streams is opened first will have precedence, and an attempt to open another stream
- `dwHDelay`: The horizontal delay factor used in the Bt848/878 `VBI_HDELAY` field (register 0xE4). For NTSC the suggested value is decimal 14. For PAL, the suggested value is decimal 32.

  In the NTSC case, if there is any possibility that the VBI stream must coexist with a decoded Closed Caption stream, it is mandatory to set dwHDelay to 14. If any other value is used, the VBI stream will fail if it is configured after the CC stream, and the CC stream will fail if it is opened after the VBI stream.
- `dwHDwords`: The number of `DWORD`s per VBI line, as used in the Bt848/878 `VBI_PKT` fields (registers 0xE0 and 0xE4). For NTSC the suggested value is 400 decimal. For PAL, the suggested value is 500 decimal.

  As with `dwHDelay`, for NTSC, if a decoded Closed Caption stream might be opened while the VBI stream is open, the `dwHDwords` value used be the recommended value of 400 – or else the second of the two streams to open will fail.
- `dwBufBytes`: The size in bytes of one VBIHDR plus its associated data buffer. The call will fail if `dwBufBytes` does not match the size the driver calculates from `dwStartLine`, `dwEndLine`, and `dwHDwords`. The driver requires that each VBIHDR + data buffer be allocated as a contiguous block. See the VbiGraph sample app, `CVbiGraphDlg::VbiInit()` function for sample code that allocates the buffers correctly and sets `dwBufBytes` to the proper value.
- `pCallback`: A pointer to the callback function in your application that the driver calls each time it has a filled buffer to return. A sample declaration of a callback function could be as follows:

  ```
  VOID CALLBACK Callback(PVOID pContext, PVBIHDR pVbiHdr);
  ```
- `pContext`: A pointer to application-defined context data. The driver passes this pointer to your application in the callback. If you do want to use this pointer, set `pContext` to NULL.

lParam2:

Set `lParam2` to `sizeof(VBIFMT)`.

MMAC_o100_VBI_CONTROL

Purpose:

Starts and stops streaming of VBI buffers.

Versions:

Supported by driver versions 1.50 and later.

lParam1:

A `DWORD` flag. Set it to one of the following two values defined in "o100ext.h":

```
#define VBI_STOP    1
#define VBI_RUN     2
```

lParam2:

Set `lParam2` to 0.

Notes:

The suggested startup sequence is to send the `MMAC_o100_VBI_CONFIG` message, then queue all your VBI buffers with the `MMAC_o100_VBI_ADD_BUFFER` message, then call `MMAC_o100_VBI_CONTROL` with `lParam1` set to `VBI_RUN`. This sequence will prevent queue underruns on startup.

At the end of the VBI session, you should send this message wit `lParam1` set to `VBI_STOP`, before closing the stream.

MMAC_o100_VBI_ADD_BUFFER

Purpose:

Sends a VBI buffer to the driver for filling.

Versions:

Supported by driver versions 1.50 and later.

lParam1:

Points to an initialized VBI_HDR structure, declared as follows in "o100ext.h":

```
typedef struct _VBIHDR {
  PUCHAR pucBuf;           /* pointer to attached VBI data             */
  DWORD  dwRev;            /* sw rev that buffer conforms to           */
  DWORD  dwFlags;          /* flags - see below                        */
  DWORD  dwSerialN;        /* serial number since open of this field   */
  DWORD  dwNQueueUnderruns; /* kernel queue underruns since last field  */
  DWORD  dwNFieldsDropped;  /* fields dropped in kernel since last field*/
}
VBIHDR, *PVBIHDR, **PPVBIHDR;
```

The first two members of VBIHDR should be set as follows when the VBIHDR and data buffer are allocated and initialized. They do not have to be set each time the buffer is sent to the driver.

`pucBuf`: Point this variable to the start of the VBI data buffer associated with this header. This data buffer must immediately follow the VBIHDR structure.

`dwRev`: Set to 0.

All other members of VBIHDR are set by the driver before it calls back to your application. They are as follows:

`dwFlags`: The driver may set any of the following bits:

```
#define VBI_ODDEVEN  0x00000001  /* 0 = odd, 1 = even field returned */
#define VBI_PURGED   0x00000002  /* 1 = buffer was purged - no data  */
#define VBI_INTERNAL 0x00000004  /* 1 = internal driver error        */
#define VBI_UNDERRUN 0x00000020  /* 1 = nonzero dwNQueueUnderruns    */
#define VBI_MISSED   0x00000040  /* 1 = nonzero dwNFieldsDropped     */
#define VBI_ERR      0x00000080  /* 1 = any of above 4 errors        */
```

These flags have meanings as follows:

`VBI_ODDEVEN`: will always be set or cleared to indicate whether the data in the buffer being returned is from the odd or even video field. If `VBI_PURGED` or `VBI_INTERNAL` is set, the buffer's VBI data is not valid and this bit is not meaningful.

`VBI_ERR`: will be set if any of the following four error bits is set. If `VBI_ERR` is not set, then the four error bits will be cleared and do not have to be examined. If `VBI_ERR` is set, then the application should examine the four error bits.

`VBI_PURGED`: The buffer is being returned from the driver without being used. This is a normal return when the stream is being closed.

VBI_INTERNAL: There was an internal driver error. The returned VBI data and VBI_ODDEVEN bit are not valid.

VBI_UNDERRUN: There were one or more queue underruns. You have not allocated enough buffers, your application is not returning buffers to the driver quickly enough, or system CPU utilization is excessively high. The returned VBI data in the current buffer is valid unless VBI_PURGED or VBI_INTERNAL is also set.

VBI_MISSED: The driver skipped one or more fields of VBI data. If this error occurs repeatedly it indicates that CPU utilization is excessive, or the system is not tuned correctly in some way. . The returned VBI data in the current buffer is valid unless VBI_PURGED or VBI_INTERNAL is also set.

dwSerialN: A sequence number for this buffer. The first buffer returned in the current VBI streaming session will be numbered 0, and subsequent buffers will be numbered in strict ascending sequential order.

dwNQueueUnderruns: If the VBI_UNDERRUN bit is set, this variable will be set to a value greater than or equal to one indicating the number of queue underruns that have occurred since the last underrun report. Note that this is a non-cumulative number. If you want to keep a total of all underruns, you must accumulate all dwNQueueUnderruns in your application.

dwNFieldsDropped: If the VBI_MISSED bit is set, this variable will be set to a value greater than or equal to one that indicating the number of missed fields that have occurred since the last missed fields report. Note that this is a non-cumulative number. If you want to keep a total of all missed fields, you must accumulate all dwNFieldsDropped in your application.

lParam2:

Set lParam2 to 0.

MMAC_o100_LOGO_CONTROL

Purpose:

Allows control of on-video logo options.

Refer to the Osprey Video Capture Driver Users' Guide, Version 1.50 or later, for a general description of the logo capability.

Refer to the "LogoApp" description for a usage example.

Versions:

Supported by driver versions 1.50 and later.

lParam1:

Points to an initialized `LOGO_CTL` structure, declared as follows in "o100ext.h":

```
typedef struct {
  DWORD dwCmd;             /* command code - see below      */
  BOOL  bEnabled;          /* logo enabled                  */
  DWORD dwLogoHt;          /* height of logo, 1x scaling    */
  DWORD dwLogoWd;          /* width  of logo, 1x scaling    */
  DWORD dwVideoHt;         /* video height for 1x scaling   */
  DWORD dwVideoWd;         /* video width  for 1x scaling   */
  DWORD dwLogoTop;         /* position of top  of logo      */
  DWORD dwLogoLeft;        /* position of left of logo      */
  DWORD dwKeyColor;        /* key color, none = 0x80000000  */
  BOOL  bEmbossed;         /* logo has embossed style       */
  WCHAR awFName[MAX_PATH]; /* path of logo .bmp file        */
}
LOGO_CTL, *PLOGO_CTL;
```

lParam2:

Set `lParam2` to `sizeof(LOGO_CTL)`.

dwCmd codes:

`dwCmd` may take on any of these five subcommand values:

```
#define LOGO_GET        4
#define LOGO_SET        6
#define LOGO_SAVE       1
#define LOGO_SET_SAVE   7
#define LOGO_RELEASE    2
```

`LOGO_GET`: Returns the driver's current logo settings. If the logo mechanism is not started, the driver reads these settings from the registry. For logo settings not present in the registry, the driver returns its internal defaults.

On input, only `dwCmd` needs to be set. On output, `dwCmd` is unchanged; all other members of `LOGO_CTL` are filled in with the driver's current logo settings.

`LOGO_SET`: Sets the driver's logo settings to the values contained in the `LOGO_CTL` structure. Logo rendering is immediately stopped, started, or modified as appropriate; if video is

not currently running, the changes take effect when video is started. These settings will override the driver's registry settings until a `LOGO_RELEASE` command is issued. The user's logo dialog will be disabled until a `LOGO_RELEASE` command is issued.

On input, all members of `LOGO_CTL` must be filled in. On output, all members are unchanged.

`LOGO_SAVE`: Writes the logo settings contained in the `LOGO_CTL` structure to the registry. No other immediate action is taken – the new values will not be recognized until the driver reads the registry again as part of its normal operations.

On input, all members of `LOGO_CTL` must be filled in. On output, all members are unchanged.

`LOGO_SET_SAVE`: Performs the actions of `LOGO_SET` and in addition saves the new logo settings to the registry.

On input, all members of `LOGO_CTL` must be filled in. On output, all members are unchanged.

`LOGO_RELEASE`: Restores the driver's logo settings to the values contained in the registry. Logo rendering is stopped, started, or modified immediately to conform to the registry settings. If video is not currently running, the changes take effect when video is started. The user's logo dialog is enabled.

On input, only `dwCmd` needs to be set. On output, all members of `LOGO_CTL` are unchanged.

If the application terminates without executing this command, the logo mechanism will be in the released state when this or any other application is restarted. In other words, the effects of `LOGO_SET` are volatile unless `LOGO_SAVE` is also issued.

LOGO_CTL members:

The members of this structure except for dwCmd are the same as the registry variables found at

```
HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device#\VcDll
```

`bEnabled`: If `TRUE`, and the other members of the structure are valid, the logo will be displayed.

`dwLogoHt`:
`dwLogoWd`: The height and width of the logo's source bitmap, as given in the .BMP file. These parameters are prior to scaling.

`dwVideoHt`:
`dwVideoWd`: Reference height and width of video frames used for scaling. If actual video frames being drawn are this size, the logo will be scaled 1x. For example, if actual video size is currently 240 high by 320 wide, and these variables are set to 240 high by 320 wide, and the logo's source bitmap is 40 pixels high and 80 pixels wide, then the logo will be rendered at 40 pixels high and 80 pixels wide. On the other hand, if actual video size is 480 high by 640 wide, then the logo will be rendered at 2x, 80 pixels high by 160 pixels wide.

`dwLogoTop`:
`dwLogoLeft`: The position on the video frame of the top left corner of the logo. These parameters are prior to scaling. That is, if `dwLogoVideoHt/Wd` are 240 and 320, actual video height and width are 480 and 640, and `dwLogoTop/Left` are 180 and 220, the actual scaled top/left positions will be 360 and 440.

`dwKeyColor`: The key color. Logo pixels that are in this key color are transparent, so that the underlying video is seen instead. The key color is in the format 0x00RRGGBB – that

is, the low byte is the blue color, the next byte is the green color, the next byte is the red color, and the high byte should always be zero. The key colors possible using the sdk can be any of the 2^24 colors allowed by the data format – they are not restricted to the four choices in the driver's user dialog. If no key color is wanted, set `dwLogoKeyColor` to 0x80000000.

`bEmbossed`: If `TRUE`, the colors of active logo pixels will be the average of the logo bitmap color and the underlying video color.

`awFName`: The full path to the logo .BMP file in Unicode format. This file must be saved in 24-bit BMP format.

MMAC_o100_CROP_CONTROL

Purpose:

Allows control of cropping options.

Refer to the Cropping User's Manual, Version 1.0 or later, for a general description of cropping.

Refer to the "CropApp" description for a usage example.

Versions:

Supported by driver versions 1.54 and later.

lParam1:

Points to an initialized `CROP_CTL` structure, declared as follows in "o100ext.h":

```
typedef struct {
  DWORD dwCmd;              /* command code -- see below     */
  BOOL  bEnabled;           /* cropping enabled              */
  DWORD dwCropXOffset;      /* X offset for cropping         */
  DWORD dwCropYOffset;      /* Y offset for cropping         */
  DWORD dwCropWidth;        /* width of cropped region       */
  DWORD dwCropHeight;       /* height of cropped region      */
  DWORD dwVideoWidth;       /* width of original image       */
  DWORD dwVideoHeight;      /* height of original image      */
}
CROP_CTL, *PCROP_CTL;
```

lParam2:

Set `lParam2` to `sizeof(CROP_CTL)`.

dwCmd codes:

`dwCmd` may take on any of these two subcommand values:

```
#define CROP_GET      1
#define CROP_SET      2
```

`CROP_GET`: Returns the driver's current cropping settings.

On input, only `dwCmd` needs to be set. On output, `dwCmd` is unchanged; all other members of `CROP_CTL` are filled in with the driver's current cropping settings.

`CROP_SET`: Sets the driver's cropping settings to the values contained in the `CROP_CTL` structure. Cropping should not be changed if video is currently running. All changes made while the video is stopped will take effect when video is started. These settings will replace the driver's registry settings.

On input, all members of `CROP_CTL` must be filled in. The cropping parameters may be modified based on hardware requirements. On output the members of `CROP_CTL` contain the actual cropping parameters being used. Any values that do not match the hardware requirements will be rounded down to the closest legal value. The table below lists the requirements for the horizontal and vertical cropped window size for each of the color formats supported by the card.

Color Format	Horizontal	Vertical
RGB32	none	even
RGB24	multiple of 4	even
RGB16,RGB15	even	even
RGB8,GRAY8	multiple of 4	even
YUY2 (4:2:2 packed)	even	even
I420 (YUV12 planar)	multiple of 16	even
YVU9 (YVU9 planar)	multiple of 16	multiple of 4

CROP_CTL members:

The members of this structure except for dwCmd are the same as the registry variables found at

`HKEY_CURRENT_USER\Software\Osprey\Osprey100\Device#\VcDll`

`bEnabled`: If `TRUE`, cropping will be enabled using the remaining parameters in the structure. If `FALSE`, cropping will be disabled and the default cropping settings will be set using the remaining parameters.

`dwCropXOffset`:
`dwCropYOffset`: The upper-left corner of the cropping box. These parameters are prior to scaling.

`dwCropWidth`:
`dwCropHeight`: The size of the cropping box. These parameters are prior to scaling.

`dwVideoWidth`:
`dwVideoHeight`: The original image size before image cropping. These values are for reference. This command may not be used to change the image size. Use the `MMAC_o100_SRC_CONTROL` message to change the video format or pixel aspect ratio of the original image.

MMAC_o100_REGISTER_GET

Purpose:

Read Bt848/878 registers directly from your application.

Versions:

Supported by driver versions 1.35 and later.

Sample Usage:

Refer to the sample application RegApp.

```
REGISTER_CTL oRegCtl;
BOOL bOk;

oRegCtl.dwReg = 4;  // the address of the Bt848 IFORM register

bOk = SendDriverMessage(
  hDrvr,
  MMAC_o100_REGISTER_GET,
  (LONG)&oRegCtl,
  sizeof(REGISTER_CTL)
);

if (!bOk)
  return FALSE;

printf("IFORM = %02X", oRegCtl.dwData);
```

lParam1:

A pointer to a `REGISTER_CTL` structure, declared in <o100ext.h> as follows:

```
typedef struct {
  DWORD dwReg;    /* byte or dword register to read or write */
  DWORD dwData;   /* byte or dword read or write data        */
}
REGISTER_CTL, *PREGISTER_CTL;
```

lParam2:

Set `lParam2` to `sizeof(REGISTER_CTL)`.

Description:

Use of these functions requires Conexant Bt848/878 documentation, not supplied with this kit, obtainable at www.conexant.com.

This message is used for reading both byte and dword registers. For byte registers, `REGISTER_CTL`'s members are used as follows:

in:	`dwReg`	= address offset of byte register - 0x000..0x0FF
	`dwData`	= unused

out: `dwReg` = unchanged
`dwData` = byte return data, zero padded

For dword registers, usage is as follows:

in: `dwReg` = address offset of dword register - 0x100..0x2FF
`dwData` = unused

out: `dwReg` = unchanged
`dwData` = dword return data

The `SendDriverMessage()` call returns nonzero if successful, zero if unsuccessful.

MMAC_o100_REGISTER_SET

Purpose:

Write Bt848/878 registers directly from your application.

Direct setting of registers in this way is generally not recommended. Osprey will not assist debug of applications that use `MMAC_o100_REGISTER_SET`.

Versions:

Supported by driver versions 1.35 and later.

Sample Usage:

Refer to the sample application RegApp.

```
REGISTER_CTL oRegCtl;
BOOL bOk;

oRegCtl.dwReg  =    4;  // the address of the Bt848 IFORM register
oRegCtl.dwData = 0x69;  // a possible value

bOk = SendDriverMessage(
  hDrvr,
  MMAC_o100_REGISTER_SET,
  (LONG)&oRegCtl,
  sizeof(REGISTER_CTL)
);

if (!bOk)
  return FALSE;
```

lParam1:

A pointer to a `REGISTER_CTL` structure. See `MMAC_o100_REGISTER_SET` for the declaration.

lParam2:

Set `lParam2` to `sizeof(REGISTER_CTL)`.

Description:

For byte registers, `REGISTER_CTL` members are used as follows:

in:	`dwReg`	= address offset of byte register - 0x000..0x0FF
	`dwData`	= byte data to write
out:	`dwReg`	= unchanged
	`dwData`	= unchanged

For dword registers, usage is as follows:

in:	`dwReg`	= address offset of dword register - 0x100..0x2FF
	`dwData`	= dword data to write

out: `dwReg` = unchanged
`dwData` = unchanged

The `SendDriverMessage()` call returns nonzero if successful, zero if unsuccessful.

MMAC_ADV_ CONTROL

Purpose:

To control advanced video processing features on Osprey cards. `MMAC_ADV_CONTROL` replaces the driver message `MMAC_O500_CONTROL` from the Osprey-500 SDK. For backwards compatibility, the SDK supports the `MMAC_O500_CONTROL` message as a synonym for `MMAC_ADV_CONTROL`.

Sample Usage:

```
OADVCTL oCtl;
BOOL    bOk;

oCtl.dwMode = 0;  // read current values
bOk = SendDriverMessage(
  hDrvr,
  MMAC_ADV_CONTROL,
  (LONG)&oCtl,
  sizeof(OADVCTL)
);

if (bOk) {
    oCtl.dwFlags |= DEINTERLACE_FILTER; // turn on filter

  oCtl.dwMode = 1;  // set new values
  bOk = SendDriverMessage(
    hDrvr,
    MMAC_ADV_CONTROL,
    (LONG)&oCtl,
    sizeof(OADVCTL)
  );
}
```

IParam1:

Points to an initialized `OADVCTL` structure, declared as follows in <o100ext.h>:

```
typedef struct {
  DWORD     dwMode;     /* 0=get values, 1 = set values */
  DWORD     dwFlags;    /* bit flags for features */
  PVOID     pReserved1; /* reserved for future use */
  PVOID     pReserved2; /* reserved for future use */
  DWORD     dwReserved1;/* reserved for future use */
  DWORD     dwReserved2;/* reserved for future use */
}
OADVCTL, *POADVCTL, **PPADVCTL;
```

The bits of `dwFlags` are defined as follows:

```
DEINTERLACE_FILTER    0x00000001L /* Enable hardware deinterlace filter */
ANALOG_INPUT          0x00000002L /* Enable analog video     */
PROGRESSIVE_SEND      0x00000004L /* Enable even/odd send */
SW_DEINTERLACE_FILTER 0x00000008L /* Enable software deinterlace filter */
```

```
FORCE_ODD_FIRST       0x00000010L  /* Force field pairing to be odd field
first */
FORCE_EVEN_FIRST      0x00000020L  /* Force field pairing to be even field
first */
```

lParam2:

Set `lParam2` to `sizeof(`OADVCTL`)`.

Description:

Use this message to control the video processing features on Osprey cards. Use the `dwMode` member of OADVCTL to specify whether the driver should return the current settings or accept new settings for each of the features.

MMAC_o100_SETTINGS_GET / SET

Purpose:

To get and set low-level driver parameters.

Versions:

Supported by driver versions 1.21 and later. These messages are obsolete. They were replaced by the `MMAC_o100_SETTINGS_GET2` and `MMAC_o100_SETTINGS_SET2` messages for driver version 1.30, which were in turn replaced by the `MMAC_o100_SRC_CONTROL` message for driver version 1.35.

Sample Usage:

See the description for `MMAC_o100SETTINGS_GET2/SET2` below.

lParam1:

A pointer to an `EXTSETTINGS` structure, declared in <o100ext.h> as follows:

```
typedef struct {
  HWND      hWnd;          /* client draw window       */
  DWORD     dwDeviceN;     /* hardware board no.       */
  DWORD     dwMux;         /* video source multiplexer */
  DWORD     dwVFmt;        /* NTSC/PAL + SQ/CCIR       */
  DWORD     dwHeight;      /* image pixel height       */
  DWORD     dwWidth;       /* image pixel width        */
  DWORD     dwColorFmt;    /* RGB32, etc               */
  DWORD     dwFlags;       /* control flags            */
  CONTROLS  oCtls[3];      /* controls for 3 sources   */
}
EXTSETTINGS, *PEXTSETTINGS, **PPEXTSETTINGS;
```

These variables are copies of the driver's internal variables. Because they are doubly obsolete they are not documented here. Contact Osprey by email if you need information about them.

lParam2:

Set `lParam2` to `sizeof(EXTSETTINGS)`.

MMAC_o100_SETTINGS_GET2 / SET2

Purpose:

To get and set low-level driver parameters.

Versions:

Supported by driver versions 1.30 and later. These messages are obsolete. They are replaced by the `MMAC_o100_SRC_CONTROL` message as of driver version 1.35.

Sample Usage:

```
EXTSETTINGS2 oExtSettings2;
BOOL bOk;

// Must initialize oExtSettings2 with which board to access before the
call.
oExtSettings2.dwDeviceN = 0;

// Get the current settings:
bOk = SendDriverMessage(
  hDrvr,
  MMAC_o100_SETTINGS_GET2,
  (LONG)&oExtSettings2,
  sizeof(EXTSETTINGS2)
);

if (!bOk)
  return FALSE;

oExtSettings2.dwMux = 2;   // a possible new setting

// Put the revised settings:
SendDriverMessage(
  hDrvr,
  MMAC_o100_SETTINGS_SET2,
  (LONG)&oExtSettings2,
  sizeof(EXTSETTINGS2)
);
```

lParam1:

A pointer to an `EXTSETTINGS2` structure, declared in <o100ext.h> as follows:

```
typedef struct {
  HWND      hWnd;          /* client draw window       */
  DWORD     dwDeviceN;     /* hardware board no.       */
  DWORD     dwMux;         /* video source multiplexer */
  DWORD     dwVFmt;        /* NTSC/PAL + SQ/CCIR       */
  DWORD     dwHeight;      /* image pixel height       */
  DWORD     dwWidth;       /* image pixel width        */
  DWORD     dwColorFmt;    /* RGB32, etc               */
  DWORD     dwFlags;       /* control flags            */
  CONTROLS  oCtls[4];      /* controls for 4 sources   */
```

```
    DWORD      dwCCFlags;       /* control flags for CC      */
    DWORD      dwDvcType;       /* type of board             */
  }
  EXTSETTINGS2, *PEXTSETTINGS2, **PPEXTSETTINGS2;
```

lParam2:

Set `lParam2` to `sizeof(EXTSETTINGS2)`.

Description:

As the example above shows, the only member to set prior to sending an `MMAC_o100_SETTINGS_GET2` message is dwDeviceN.

Due to the subtlety of the internal settings, before issuing an `MMAC_o100_SETTINGS_SET2` message, it is effectively mandatory to get the current settings with `MMAC_o100_SETTINGS_GET2`, then modify specific variables, then put back the modified settings.

`EXTSETTINGS2` exposes some variables that should only be modified by Video for Windows commands – most notably, dwHeight, dwWidth, and dwColorFmt. These should be set by VfW's `DVM_FORMAT` message. If these variables are set directly, the driver may not properly handle all of the required side effects.

Most of the bits of `dwFlags` are private to the driver. The only bits that might provide useful information to an application are the following, defined in <o100ext.h>. These bits indicate which overlay mode the video capture driver is using:

```
#define DDRAW_PRI   0x00000001  /* ddrawing to primary   surface  */
#define DDRAW_2ND   0x00000002  /* ddrawing to secondary surface  */
#define DIBDRAW     0x00000004  /* dibdraw overlaying             */
#define OVERLAY     0x00000008  /* overlaying video               */
```

`EXTSETTINGS2` overcomes deficiencies in the original EXTSETTINGS structure:

1. It allows setting of brightness, contrast, etc. for all possible four video inputs.
2. It allows direct control of the Closed Caption control flags.
3. It allows an application to access low-level information about characteristics of the boards installed in the system.

`EXTSETTINGS2` and its messages have a severe problem which make it very desirable to use the newer `MMAC_o100_SRC_CONTROL` instead in nearly all cases: The video controls array `oCtls[]` is indexed by the physical video input multiplexer settings. Different Osprey-100 card versions have different numbers and arrangements of connectors. `MMAC_o100_SRC_CONTROL` uses logical rather than physical settings that are independent of the specific board type and are designed to work with all present and future Osprey-100 variants.

Structures

The SDK's structures are all associated with specific functions and messages, and so they are described as part of the descriptions of those functions and messages. The following is a cross-reference to these descriptions:

- `O100_OPEN` – see `OpenDriver()`
- `SRCCTL` – see `MMAC_o100_SRC_CONTROL`
- `CONTROLS` – see `MMAC_o100_SRC_CONTROL`
- `CC_CALLBACK_CTL` – see `MMAC_o100_CC_CONTROL`
- `REGISTER_CTL` – see `MMAC_o100_REGISTER_GET`
- `CAPS_QUERY` – see `MMAC_CAPS_QUERY`
- `EXTSETTINGS` – see `MMAC_o100_SETTINGS_GET/SET`
- `EXTSETTINGS2` – see `MMAC_o100_SETTINGS_GET2/SET2`
- `LOGO_CTL` – see `MMAC_o100_LOGO_CONTROL`
- `CROP_CTL` – see `MMAC_o100_CROP_CONTROL`
- `OADVCTL` – see `MMAC_ADV_CONTROL`

These structures are declared in the SDK include files <o100ext.h> and <mmacdrv.h>.