

Programmer's Manual

MM/104 Multimedia Board

Ver. 1.0 – 11. Sep. 1998.

Copyright Notice :

Copyright © 1998, INSIDE Technology A/S, ALL RIGHTS RESERVED.

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, with out the express written permission of INSIDE Technology A/S.

Trademark Acknowledgement :

Brand and product names are trademarks or registered trademarks of their respective owners.

Disclaimer :

INSIDE Technology A/S reserves the right to make changes, without notice, to any product, including circuits and/or software described or contained in this manual in order to improve design and/or performance. INSIDE Technology assumes no responsibility or liability for the use of the described product(s), conveys no license or title under any patent, copyright, or mask work rights to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are described in this manual are for illustration purposes only. INSIDE Technology A/S makes no representation or warranty that such application will be suitable for the specified use without further testing or modification.

The following code is sample code created by INSIDE Technology. This sample code is provided to you solely for the purpose of assisting you in the development of your applications. The code is provided "AS IS", without warranty of any kind. INSIDE Technology shall not be liable for any damages arising out of your use of the sample code.

Life Support Policy

INSIDE Technology's PRODUCTS ARE NOT FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT EXPRESS WRITTEN APPROVAL OF THE GENERAL MANAGER OF INSIDE Technology A/S.

As used herein :

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into body, or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labelling, can be reasonably expected to result in significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

1	INTRODUCTION	1
2	REQUIREMENTS.	2
3	SNIPPETS.	2
3.1	CREATING A CAPTURE WINDOW.....	4
3.2	HARDWARE-OVERLAY.	4
3.3	SOFTWARE PREVIEW.	5
3.4	SINGLE FRAME CAPTURE.	5
3.5	DIALOGS.	6
3.6	CAPTURING TO A FILE.	6
3.7	USING FRAME CALLBACKS.....	7
3.8	TIPS.	9

Document revision history.

Revision	Date	By	Comment
0.1	July 1998	DHO	Initial version
1.0	Sep 1998	DHO	Revised version

1 Introduction

This document describes how to use a selected subset of the Video for Windows (VFW) API with INSIDE technology's MM/104 card. Specifically this document shows how to use the video-acquisition part of the MM/104 board under Windows 95/98, using C. The purpose of this document is to help developers getting started using VFW - it is not meant as a reference. It is recommended to obtain a proper reference for the VFW API e.g. from MSDN:

<http://premium.microsoft.com/msdn> in the "SDK Documentation/Platform SDK/Graphics and Multimedia Services/Video for Windows/Video Capture" section. Microsoft Visual C++ (version 5.0) also includes an online manual to VFW (must be installed explicitly). The users targeted with this reference are system programmers with a good understanding of the Windows programming environment and terminology. The examples in this document are available electronically, refer to the examples directory on the MM/104 driver diskette.

For additional information about INSIDE Technology A/S and our products please visit our homepage:

Denmark - <http://www.inside.dk> (International Headquarters)

USA – <http://www.inside-us.com>

2 Requirements.

In order to use the following snippets, a capture-device and a Windows-compatible C compiler are required. For compiling the samples with Microsoft Visual C++ version 5.0, the attached MAKE.BAT file can be used. Note that the VC environment variables must be properly set before you can use the script (i.e. run the VCVARS32.BAT file in the Visual C++ binary directory).

3 Snippets.

The following snippets demonstrate how to use some of the VFW API. Most of the functionality in VFW is implemented as messages sent to the capture-driver using SendMessage. In fact, most applications only use the functions capCreateCaptureWindow and capGetDriverDescription. VFW also defines a large number of macros that serves as wrappers around the messages. The following snippets are made as compact and readable as possible. Many of the bells and whistles are ignored. The following examples all assumes that the driver number 0 is present, not in use by an other application. The “Frame Callback” example assumes that the driver runs in 16 bit mode in 320x240 pixels. The other examples do not rely on a specific resolution or colordepth.

The following snippets are all modifications to a simple skeleton like this:

```
#include <windows.h>
#include <vfw.h>

HWND mainwindow;
HWND capturewindow;

LRESULT CALLBACK windowfunc(HWND hwnd, UINT msg,
                             WPARAM wparam, LPARAM lparam)
{
    switch(msg)
    {
        case WM_CLOSE:
            DestroyWindow(mainwindow);
            return 0;

        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;

        default:
            return DefWindowProc(hwnd, msg, wparam, lparam);
    }

    return 0;
}
```

(Continued on next page)

```

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpszCmdLine, int nCmdShow)
{
    MSG msg;
    WNDCLASS wc;
    HINSTANCE hinst;
    lpszCmdLine=lpszCmdLine;

    if (!hPrevInstance)
    {
        wc.style = 0;
        wc.lpfnWndProc = (WNDPROC) windowfunc;
        wc.cbClsExtra = 0;
        wc.cbWndExtra = 0;
        wc.hInstance = hInstance;
        wc.hIcon = LoadIcon((HINSTANCE) NULL,
                            IDI_APPLICATION);
        wc.hCursor = LoadCursor((HINSTANCE) NULL, IDC_ARROW);
        wc.hbrBackground = GetStockObject(WHITE_BRUSH);
        wc.lpszMenuName = "MainMenu";
        wc.lpszClassName = "captureclass";

        if (!RegisterClass(&wc))
            return FALSE;
    }

    hinst = hInstance;

    mainwindow = CreateWindow("captureclass", "Sample",
                             WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
                             CW_USEDEFAULT, CW_USEDEFAULT, (HWND) NULL,
                             (HMENU) NULL, hinst, (LPVOID) NULL);

    if (!mainwindow)
        return FALSE;

    ShowWindow(mainwindow, nCmdShow);
    UpdateWindow(mainwindow);

    while (GetMessage(&msg, (HWND) NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return msg.wParam;
}

```

3.1 Creating a capture window.

Before any of the VFW API can be used, a capture window must be created. The capture window serves two purposes: It is used to preview captured frames and to relay messages to the capture driver. The capture window is created with the function `capCreateCaptureWindow`. It takes 8 arguments and returns a window handle to the created capture-window (or NULL in case of failure), for example:

```
HWND capturewindow, mainwindow;

capturewindow = capCreateCaptureWindow("INSIDE TV",
                                       WS_CHILD | WS_VISIBLE,
                                       0, 0, 160, 120, mainwindow, 0);
```

Before the capture window can relay messages to a capture driver, it has to be connected to one. This is done with the macro `capDriverConnect` (`capDriverConnect` is really a wrapper to the `WM_CAP_DRIVER_CONNECT` message). This snippet connects `capturewindow` with the first capturedriver (number 0):

```
capDriverConnect(capturewindow, 0);
```

The capture window is now created and connected to a driver. Note that all VFW resources must be explicitly freed. The macro `capDriverDisconnect` disconnects the window from the driver. If disconnection of the driver fails upon exit, reconnection to the driver may not be possible before you have rebooted the system. The macro is used like this:

```
capDriverDisconnect(capturewindow);
```

3.2 Hardware-overlay.

Hardware overlaying uses hardware to render the video input directly into the framebuffer. As it doesn't use any CPU resources it is fast and has no impact (in terms of CPU-usage) on other applications. The macro `capOverlay` creates an overlay. It is used like this:

```
capOverlay(capturewindow, TRUE);
```

Note that the overlay is placed relative to the framebuffer not to the parent window. This means that the overlay does not follow the window when it is moved. To correct this the following lines are added to the window function:

```
case WM_MOVE:
    capOverlay(capturewindow, TRUE);
    return 0;
```

A listing is available in `overlay.c`

3.3 Software Preview.

Software preview resembles Hardware overlay as they both renders video input on the screen. Software preview uses the hardware on the capture device to digitise and transfer frames to buffers in memory. The CPU is then used to render the frames on screen. This method is slower than hardware overlay and generally eats away a lot of CPU cycles.

The macro `capPreview` is used to enable or disable software preview. It can be used together with `capPreviewRate` and `capPreviewScale`. `capPreviewRate` requests a given framerate and `capPreviewScale` enables or disables scaling and stretching of the capture window.

The macros are used like this:

```
capPreview(capturewindow, TRUE);
capPreviewScale(capturewindow, TRUE);
capPreviewRate(capturewindow, 250);
```

First preview is turned on, then scaling is turned on and at last the requested framerate is set to 250 msec/frame.

Note that when resizing the window the size of the `capturewindow` needs to be set manually. These lines adjusts the size of the `capturewindow` each time the main window is resized:

```
case WM_SIZE:
    SetWindowPos(capturewindow, NULL, 0,0,
                LOWORD(lparam), HIWORD(lparam),
                SWP_NOOWNERZORDER | SWP_NOMOVE);
    return 0;
```

A listing is available in `preview.c`

3.4 Single Frame Capture.

If you wish to grab a single frame you can use the macro `capGrabFrame`. In a demonstration `capGrabFrame` doesn't do well on its own. But by coupling it with `capEditCopy` you get the effect of a poor mans still-camera. `capEditCopy` copies the content of the capture window to the clipboard. The following lines are added to the overlay example:

```
case WM_MOUSEACTIVATE:
    capGrabFrame(capturewindow);
    capEditCopy(capturewindow);
    capOverlay(capturewindow, TRUE);
    return 0;
```

When the program starts the capture window is overlaying the video input. When the user clicks the overlaid window its contents are copied to the clipboard. While the program has a few sharp edges, it is not totally useless. The exercise of refinement is left to the user.

A listing is available in single.c.

3.5 Dialogs.

The driver defines up to four dialogs which lets the user set attributes in the driver. As it is the driver itself that defines the dialogs, their appearance will differ from driver to driver. A driver might only define a subset of the dialogs. You can use the capDriverGetCaps to query the capabilities of the driver (i.e. which dialogs are supported).

Following is a list of the macros that display the dialogs and a brief description of the dialogs (as they appear when running the Chips & Tech driver).

Macro name	Description
capDlgVideoFormat	Select the resolution and color depth in which the driver should operate.
capDlgVideoDisplay	Set saturation, brightness, contrast in overlay-mode.
capDlgVideoSource	Select hue, video connector, input format (composite/S-video) and standard (NTSC/PAL) in capture.
capDlgVideoCompression	Select compression format, quality and frame-rate.

A listing is available in dialogs.c.

3.6 Capturing to a file.

The simplest way to capture multiple frames is by using the macro capCaptureSequence. The Frames captured with capCaptureSequence is saved to disk directly. By default the file "c:\capture.avi" is used but the macro capFileSetCaptureFile can be used to set the destination file of capCaptureSequence. The macros are used like this:

```
capFileSetCaptureFile(capturewindow, "myfile.avi");
capCaptureSequence(capturewindow);
```

By default the driver stops capturing when the user press the escape key or click in the capture window with either mouse buttons. The mousepointer changes to hourglass whenever the pointer is inside the window while the capture is on. This is because `capCaptureSequence` does not return before the capture is complete. To change this `capCaptureGetSetup` and `capCaptureSetSetup` can be used to set the `fYield` flag for example:

```
CAPTUREPARMS setup;
capCaptureGetSetup(capturewindow, &setup, sizeof(setup));
setup.fYield=TRUE;
capCaptureSetSetup(capturewindow, &setup, sizeof(setup));
```

This causes the driver to spawn a working thread when capturing, thus returning from `capCaptureSequence` immediately.

A listing is available in `capture.c` and `capture2.c`.

3.7 Using Frame callbacks.

To process the frames yourself instead of having the driver storing them in a .AVI-file you can use `capCaptureSequenceNoFile` to grab frames, for example:

```
capCaptureSequenceNoFile(capturewindow);
```

When calling `capCaptureSequenceNoFile` like this, you might notice a performance decrease which indicates that something is going on. The capture-device is capturing and transferring frames to memory buffers, but they are not available to you. To access the frames you will have to register a callback function, for example:

```
LRESULT videohandler (HWND window, LPVIDEOHDR header)
{
    return 0;
}

capSetCallbackOnVideoStream(capturewindow, videohandler);
```

This causes the videohandler to be called for each frame captured. No description or even a declaration of `LPVIDEOHDR` (or `VIDEOHDR`) is available in the online manuals of Visual C++ v. 5. The declaration of `VIDEOHDR` as it appears in `VFW.H` is as follows:

```

typedef struct videohdr_tag {
    LPBYTE      lpData;          /* pointer to locked
                                data buffer */
    DWORD      dwBufferLength;  /* Length of data buffer */
    DWORD      dwBytesUsed;     /* Bytes actually used */
    DWORD      dwTimeCaptured; /* Milliseconds from start
                                of stream */
    DWORD      dwUser;         /* for client's use */
    DWORD      dwFlags;       /* assorted flags
                                (see defines)*/
    DWORD      dwReserved[4];  /* reserved for driver */
} VIDEOHDR, NEAR *PVIDEOHDR, FAR *LPVIDEOHDR;

/* dwFlags field of VIDEOHDR */
#define VHDR_DONE      0x00000001 /* Done bit */
#define VHDR_PREPARED  0x00000002 /* Set if this header
                                has been prepared */
#define VHDR_INQUEUE  0x00000004 /* Reserved for driver */
#define VHDR_KEYFRAME 0x00000008 /* Key Frame */

```

Most fields should be self-explanatory. lpData is a pointer to the raw frame. The format of the data varies with the colordepth and resolution. When the device operates in 16 bits per pixel, the pixels are encoded as 15 bit RGB values like this:

Bit #	15	14..10	9..5	4..0
Meaning	Unused	Red	Green	Blue

The pixels are organised in the "normal" line-wise fashion. Thus, the first pixel is the upper left corner and the last is the lower right corner.

The meaning of dwUser is not quite clear. Its contents might be that set with the capSetUserData macro. Use with caution.

capCaptureNoFile does not display the captured frames by itself so I use the DIB (Device Independent Bitmap) interface to display the frames in the videohandler function. The following lines show the basic use of DIB:

```

HDRAWDIB dibdc=DrawDibOpen();
DrawDibBegin(dibdc,NULL, 200,200, &bitmapheader,
             320, 240, DDF_BACKGROUNDPALE);
DrawDibDraw(dibdc, dc, 0,0,320,240,
            &bitmapheader, bitmapdata, 0,0, 320, 240, 0);
DrawDibClose(dibdc);

```

Note that the yield flag in this example is set true, as capCaptureNoFile would not return otherwise.

When capturing has been done `capCaptureStop` and `capSetCallbackOnVideoStream` should be used to clean up properly, for example:

```
capCaptureStop(capturewindow);
capSetCallbackOnVideoStream(capturewindow, NULL);
```

Calling `capSetCallbackOnVideoStream` with `NULL` disables the current handler (actually it restores the default "do-nothing" handler).

A listing is available in `callback.c`.

3.8 Tips.

When debugging programs you might experience that the capture driver gets locked if an application using it crashes, with a reboot as the only way to unlock it. This is because all resources associated with the VFW system are global and not tied to any one process. The operating-system therefore does not implicitly free the VFW resources (as one could assume). There are two lessons to learn from this: You should make sure that any and all resources are freed before exiting, and your program should not crash before it has done so (if at all).

In theory both parts are trivial, but having to reboot the system if the program crashes, can make debugging very frustrating. However there is a way to make sure that a program does not exit before it has freed its resources. By using an exceptionhandler, you can catch and handle exceptions that would otherwise kill the program without further action. The following lines show how to install an exception handler (a listing is available in `exception.c`).

```
LONG exceptionhandler (LPEXCEPTION_POINTERS arg)
{
    if(capturewindow)
        capDriverDisconnect(capturewindow);
    return EXCEPTION_CONTINUE_SEARCH;
}
```

```
SetUnhandledExceptionFilter(exceptionhandler);
```

<Operations to be handled/protected>

```
SetUnhandledExceptionFilter(NULL);
```

After the driver is freed `EXCEPTION_CONTINUE_SEARCH` is returned. This tells the system to take the default action (to do what it would have done if there were no handler installed). You can return `EXCEPTION_CONTINUE_EXECUTION` to tell the system that you have handled the condition raising the exception, and the program is free to continue executing. This is not recommended though.

A listing is available in `trap.c`.