Oh Joy! Building A Joystick Component

by Ken Otto

This article demonstrates the construction of a joystick component, using the services of MMSYSTEM.DLL, the multimedia API of both Windows 3.1 and Windows 95.

The joystick is an analog input device which provides absolute position information to your application. This information can tell you not only which direction your joystick is pointing, but how far. Several devices can utilize joystick services, including touch screens, digitizing tablets and light pens. Note, however, that these devices are not considered a replacement for the mouse.

The driver for the IBM Game Adapter (IBMJOY.DRV) supports two 2-axis joysticks with two buttons or one 3-axis joystick with 4 buttons and can be found on CompuServe in the WINSHARE forum under *General Win Utils*, the filename is IBMJOY.ZIP.

There are two methods for receiving joystick information. First, we could process joystick messages from a 'captured' joystick. A captured joystick will broadcast messages when the joystick is moved or a button is pressed. This could provide us a resolution of around 1 millisecond. Alternatively, we can poll the joystick directly.

Since our joystick component will be a descendant of type `TTimer`, the direct polling method will be a nice fit for our new component. The `TTimer` will provide us a maximum resolution of about 55 milliseconds. The joystick services we will utilize include:

- `joyGetNumDevs`: Returns the number of joysticks supported by the joystick services.
- `joyGetDevCaps`: Returns joystick capabilities.
- `joyGetPos`: Returns joystick position and button information.
- `joySetCalibration`: This one is undocumented and returns calibration values from the SYSTEM.INI file.

The component we will build will return information for one 2-D primary joystick, defined as `JOYSTICKID1`, and two buttons. All the definitions for our interface to MMSYSTEM.DLL can be found in the DELPHI\DOC\MMSYSTEM.INT file, except for our undocumented use of `joySetCalibration`, which we must define ourselves. Extending this component to process a second joystick or the third dimension of the joystick would be a trivial matter. One important note is that MMSYSTEM definitions can't be found in Delphi through the IDE help menu. These definitions reside in a separate help file: DELPHI\BIN\MMSYSTEM.HLP.

Component Construction

We will derive our component, called `TJoyStick`, from `TTimer`. Note that by default the timer `Enabled` property is `True`. We don't want our component to return joystick information until the component is initialized and we have determined the joystick's presence and capabilities. Therefore we override the default setting of `True` to `False`. You will need to add `MMSystem` to your `USES` clause (see Listing 1).

The first method we call in our component is `InitJoy`, which will prepare our joystick for use. The function `joyGetNumDevs` will return the number of joysticks the system can support to the object field `FNumOfJoysticks`:

```
FNumOfJoysticks :=
  joyGetNumDevs;
```

We then call `joyGetDevCaps` for the primary joystick, which returns the joystick capabilities into the `TJoyCaps` data structure:

```
joyGetDevCaps(JOYSTICKID1,
 @FJoyCaps1,
 SizeOf(FJoyCaps1));
```

Now we must determine if the joystick is plugged in and ready to use. Calling `joyGetPos` will query the joystick and determine if it is ready. If no error is detected, joystick position information will be returned in the object variable `FJoyInfo1`:

```
if joyGetPos(JOYSTICKID1,
  @FJoyInfo1) =
  JOYERR_NOERROR then
  FJoystick1Ready := True
  {joystick is plugged in
   and ready}
else
  FJoystick1Ready := False;
  {driver present, no joystick
   plugged in}
```

If our call to `InitJoy` is successful, we can then load the calibration data. During Windows startup, MMSYSTEM.DLL loads the joystick driver, however joystick calibration data is not loaded. This information is stored in SYSTEM.INI:

```
[joystick.drv]
JoyCal0=0d4c 0094 0eaf 00b3 0000 0001
JoyCal1=0000 0001 0000 0001 0000 0001
```

The trick here is to load the primary joystick values (`JoyCal0`) into the following data structure (object field `FJoyCalibrate`):

```
{ This is an undocumented
  structure so we must declare
  it in our component }
TJoyCalibrate = record
  XBase  : Word;
  XDelta : Word;
  YBase  : Word;
  YDelta : Word;
  ZBase  : Word;
  ZDelta : Word;
end; { TJoyCalibrate }
```

➤ *Listing 1: Full source code for the TJoyStick component, which is also included on this month's disk along with the example program discussed in the article. Of course, you will need a joystick and the driver in order to make use of this code!*

```
unit Joystick;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls, MMSystem, IniFiles;
type
  TReturnLevel = (Level1,Level2,Level3);
  {This is an undocumented structure so we must declare it}
  TJoyCalibrate = record
    XBase      : Word;
    XDelta     : Word;
    YBase      : Word;
    YDelta     : Word;
    ZBase      : Word;
    ZDelta     : Word;
  end; {TJoyCalibrate}
  TJoyStick = class(TTimer)
  private { Private declarations }
    FJoyInfo1       : TJoyInfo;
    FJoyCaps1       : TJoyCaps;
    FJoyCalibrate   : TJoyCalibrate;
    FJoy1B1         : Word;
    FJoy1B2         : Word;
    FNumOfJoySticks : Word;
    FJoystick1Ready : Boolean;
    FReturnLevel    : TReturnLevel;
  protected { Protected declarations }
  public { Public declarations }
    constructor Create(AOwner : TComponent); override;
    property NumOfJoySticks : Word read FNumOfJoySticks;
    property Joy1X : Word read FJoyInfo1.wXPos;
    property Joy1Y : Word read FJoyInfo1.wYPos;
    property Joy1Z : Word read FJoyInfo1.wZPos;
    property Joy1B1 : Word read FJoy1B1;
    property Joy1B2 : Word read FJoy1B2;
    function InitJoy : Boolean;
    procedure ProcessJoystickInfo;
    procedure CalibrateJoyStick;
  published { Published declarations }
    property ReturnLevel : TReturnLevel
      read FReturnLevel write FReturnLevel;
  end; {TJoyStick}
{undocumented use}
function joySetCalibration(JoyID : Word; XBase : PWord;
  XDelta : PWord; YBase : PWord; YDelta : PWord;
  ZBase : PWord; ZDelta : PWord): Word;
procedure Register;
implementation
{undocumented use}
function joySetCalibration(JoyID : Word; XBase : PWord;
  XDelta : PWord; YBase : PWord; YDelta : PWord; ZBase :
  PWord; ZDelta : PWord): Word;  external 'MMSYSTEM';
procedure TJoystick.ProcessJoystickInfo;
begin {process info for 1 joystick}
  case FReturnLevel of
    Level1 : begin {raw data from Joystick}
      joyGetPos(JOYSTICKID1, @FJoyInfo1);
      FJoy1B1 :=  FJoyInfo1.wButtons and 1;
      FJoy1B2 := (FJoyInfo1.wButtons and 2) shr 1;
    end; {Level1}
    Level2 : begin {data SHR 11, values 0-31}
      joyGetPos(JOYSTICKID1, @FJoyInfo1);
      FJoyInfo1.wXPos := FJoyInfo1.wXPos SHR 11;
      FJoyInfo1.wYPos := FJoyInfo1.wYPos SHR 11;
      FJoyInfo1.wZPos := FJoyInfo1.wZPos SHR 11;
      FJoy1B1 :=  FJoyInfo1.wButtons and 1;
      FJoy1B2 := (FJoyInfo1.wButtons and 2) shr 1;
    end; {Level2}
    Level3 : begin {data SHR 13, values 0-7}
      joyGetPos(JOYSTICKID1, @FJoyInfo1);
      FJoyInfo1.wXPos := FJoyInfo1.wXPos SHR 13;
      FJoyInfo1.wYPos := FJoyInfo1.wYPos SHR 13;
      FJoyInfo1.wZPos := FJoyInfo1.wZPos SHR 13;
      FJoy1B1 :=  FJoyInfo1.wButtons and 1;
      FJoy1B2 := (FJoyInfo1.wButtons and 2) shr 1;
    end; {Level3}
  end; {case}
end; {TJoystick.ProcessJoystickInfo}
function TJoystick.InitJoy : Boolean;
begin
  FNumOfJoysticks := joyGetNumDevs;
  if FNumOfJoysticks in [1,2] then begin
    {system can support 1 or 2 joysticks}
    Result := True;
    joyGetDevCaps(JOYSTICKID1, @FJoyCaps1,
      SizeOf(FJoyCaps1));
      if joyGetPos(JOYSTICKID1, @FJoyInfo1) =
        JOYERR_NOERROR then
        FJoystick1Ready := True
        {joystick is plugged in and ready}
      else
        FJoystick1Ready := False;
        {driver present, no joystick plugged in}
  end else
    Result := False; {no joystick driver is present}
end; {TJoystick.InitJoy}
constructor TJoyStick.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  Enabled := False; {override default setting of True}
  FReturnLevel := Level1; {raw values from Joystick}
end; {TJoyStick.Create}
{No error checking is performed, since the values are
 coming from SYSTEM.INI and not a user. The string
 passed in will have a length of 4. It would be easy
 to add necessary error checking if needed}
function HexStrToInt(var HexStr : String) : Word;
var
  X       : ShortInt;
  TempInt : Word;
  Total   : Word;
begin
  HexStr := AnsiLowerCase(HexStr); {convert to lower case}
  Total := 0;
  for X :=  1 to Length(HexStr) do begin
    if (HexStr[X] >= 'a') then { value is between a-f }
      TempInt := Ord(HexStr[X]) - Ord('a') + 10
    else { value is between 0-9 }
      TempInt := Ord(HexStr[X]) - Ord('0');
    Total := (Total shl 4) + TempInt;
  end; {for}
  Result := Total;
end; {HexStrToInt}
procedure TJoyStick.CalibrateJoystick;
var
  Joy1ConfigStr : String[29];
  Joy2ConfigStr : String[29];
  SysIni : TIniFile;
  HexStr : String[4];
begin
  SysIni := TIniFile.Create('SYSTEM.INI');
  Joy1ConfigStr :=
    SysIni.ReadString('joystick.drv', 'JoyCal0', '');
  Joy2ConfigStr :=
    SysIni.ReadString('joystick.drv', 'JoyCal1', '');
  Move(Joy1ConfigStr[1], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.XBase := HexStrToInt(HexStr);
  Move(Joy1ConfigStr[6], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.XDelta := HexStrToInt(HexStr);
  Move(Joy1ConfigStr[11], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.YBase := HexStrToInt(HexStr);
  Move(Joy1ConfigStr[16], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.YDelta := HexStrToInt(HexStr);
  Move(Joy1ConfigStr[21], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.ZBase := HexStrToInt(HexStr);
  Move(Joy1ConfigStr[26], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.ZDelta := HexStrToInt(HexStr);
  joySetCalibration(0, @FJoyCalibrate.XBase,
    @FJoyCalibrate.XDelta, @FJoyCalibrate.YBase,
    @FJoyCalibrate.YDelta, @FJoyCalibrate.ZBase,
    @FJoyCalibrate.ZDelta);
  Move(Joy2ConfigStr[1], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.XBase := HexStrToInt(HexStr);
  Move(Joy2ConfigStr[6], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.XDelta := HexStrToInt(HexStr);
  Move(Joy2ConfigStr[11], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.YBase := HexStrToInt(HexStr);
  Move(Joy2ConfigStr[16], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.YDelta := HexStrToInt(HexStr);
  Move(Joy2ConfigStr[21], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.ZBase := HexStrToInt(HexStr);
  Move(Joy2ConfigStr[26], HexStr[1], 4);
  HexStr[0] := Chr(4);
  FJoyCalibrate.ZDelta := HexStrToInt(HexStr);
  joySetCalibration(1, @FJoyCalibrate.XBase,
    @FJoyCalibrate.XDelta, @FJoyCalibrate.YBase,
    @FJoyCalibrate.YDelta, @FJoyCalibrate.ZBase,
    @FJoyCalibrate.ZDelta);
end; {TJoyStick.CalibrateJoystick}
procedure Register;
begin
  RegisterComponents('Develop', [TJoyStick]);
end;
end. {Joystick}
```

The values we are loading are hexadecimal strings. The function to convert these hexadecimal string values to integer values is called HexStrToInt. Now we call:

```
joySetCalibration(1,
  @FJoyCalibrate.XBase,
  @FJoyCalibrate.XDelta,
  @FJoyCalibrate.YBase,
  @FJoyCalibrate.YDelta,
  @FJoyCalibrate.ZBase,
  @FJoyCalibrate.ZDelta);
```

and our joystick is calibrated with the system values.

Our component includes the published property ReturnLevel. The call to joyGetPos will return values in the range of 0 to 65535, in increments of the associated delta value found in TJoyCalibrate. This may perhaps be more resolution than the component user may necessarily require.

By shifting bits of a word, we can return varying resolutions. If we shift a word to the right 11 places, we will return values between 0 and 31 in each axis. Our center position should be approximately 15. Moving the joystick to the left, the X axis approaches 0, to the right it will approach 31.

Consider a game with the joystick controlling a man and this man can walk, trot and run in both directions of the X axis. Therefore returning values between 0 and 31 may be more appropriate than values 0 to 65535.

Once the programmer places the ProcessJoystickInfo method into the OnTimer event of the joystick component, the method will poll the current position of the joystick and if a ReturnLevel of 2 or 3 is selected, shift the bits of the joystick's current position by the appropriate amount. For example, ReturnLevel of 2 looks like this:

```
Level2 :
  begin
    {data SHR 11, values 0-31}
    joyGetPos(JOYSTICKID1,
      @FJoyInfo1);
      {joystick is polled}
    FJoyInfo1.wXPos :=
      FJoyInfo1.wXPos SHR 11;
      {the bits are shifted}
    FJoyInfo1.wYPos :=
      FJoyInfo1.wYPos SHR 11;
    FJoy1B1 :=
      FJoyInfo1.wButtons and 1;
    FJoy1B2 :=
      (FJoyInfo1.wButtons
       and 2) shr 1;
  end; {Level2}
```

Also note that button 1 and button 2 information is returned as 0 (off or up) and 1 (on or pressed).

➤ *Listing 2: Sample application*

```
unit Main;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
Graphics, Controls,
  Forms, Dialogs, ExtCtrls, Joystick, MMSystem;
type
  TfrmMain = class(TForm)
    JoyStick1: TJoyStick;
    procedure JoyStick1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private { Private declarations }
    FPoint     : TPoint; {stores the cursor position}
    {handle to a memory location for the wave file}
    FWavFile   : THandle;
    {pointer to the wave file in memory}
    FWavPtr    : Pointer;
  public  { Public declarations }
  end;
var frmMain: TfrmMain;

implementation
{$R *.DFM}

procedure TfrmMain.JoyStick1Timer(Sender: TObject);
begin
  Joystick1.ProcessJoystickInfo; {poll the joystick}
  {get the cursors current position}
  GetCursorPos(FPoint);
  {test the directional info from the joystick}
  if Joystick1.Joy1X < 2 then
    {and set the new cursor coordinates}
    FPoint.X := FPoint.X - 10
  else if Joystick1.Joy1X > 5 then
    FPoint.X := FPoint.X + 10;
  if Joystick1.Joy1Y < 2 then
    FPoint.Y := FPoint.Y - 10
  else if Joystick1.Joy1Y > 5 then
    FPoint.Y := FPoint.Y + 10;
  {place the cursor at the new coordinates}
  SetCursorPos(FPoint.X, FPoint.Y);
  if Joystick1.Joy1B1 = 1 then begin
    {if button is pressed, change the cursor}
    if Screen.Cursor = crMultiDrag then
      Screen.Cursor := crDefault
    else
      Screen.Cursor :=  Pred(Screen.Cursor);
  end;
  if Joystick1.Joy1B2 = 1 then
    {if button is pressed, play the wave file}
    sndPlaySound(FWavPtr, SND_ASYNC or SND_MEMORY);
end;

procedure TfrmMain.FormCreate(Sender: TObject);
var
  DingWav    : PChar;
  WinDirSize : Integer;
  WavSize    : LongInt;
  WavPath    : array[0..144] of Char;
  WavHandle  : THandle;
begin
  DingWav := '\ding.wav'; {the wave file we want to play}
  FillChar(WavPath, SizeOf(WavPath), #0); {init buffer}
  if Joystick1.InitJoy then begin
    {attempt to initialize the joystick}
    Joystick1.CalibrateJoystick; {calibrate the joystick}
    {enable joystick component}
    JoyStick1.Enabled := True;
  end;
  {find Win dir}
  WinDirSize :=
    GetWindowsDirectory(WavPath, SizeOf(WavPath));
  {append the wave file}
  Move(DingWav[0], WavPath[WinDirSize],
    StrLen(DingWav));
  WavHandle := _lopen(WavPath, OF_READ); {open the file}
  {determine file size}
  WavSize := _llseek(WavHandle, 0, 2);
  {reset the file to the begining}
  _llseek(WavHandle, 0, 0);
  {alloc memory}
  FWavFile :=
    GlobalAlloc( GMEM_MOVEABLE + GMEM_SHARE, WavSize);
  {get a pointer to the allocated memory}
  FWavPtr :=  GlobalLock(FWavFile);
  {copy wave file to memory location}
  _lread(WavHandle, FWavPtr, WavSize);
  _lclose(WavHandle); {close the file}
end;

procedure TfrmMain.FormDestroy(Sender: TObject);
begin
  GlobalUnlock(FWavFile); {remove the lock on the memory}
  GlobalFree(FWavFile); {free the memory}
end;

end.
```

Sample Application

Putting this joystick component to work in a sample application will demonstrate its capabilities. Let's call our demo JoyDemo. We want this application to be simple enough to allow quick construction, yet still exercise all of the features in the joystick component. JoyDemo will move a cursor around the screen. By pushing button 1 on the joystick, we will cycle through all of the pre-defined cursors in Windows. Pushing button 2 will sound the Windows DING.WAV file.

First we must place `MMSystem` into the `USES` clause. Since we are going to play a sound file, DING.WAV, we will use the function `sndPlaySound`. Although `MMSystem` was declared in our joystick component, it is not within scope for JoyDemo. The `FormCreate` event of JoyDemo will initialize the joystick. If successful, it will then calibrate the joystick and enable it. We then retrieve the Windows directory using `GetWindowsDirectory` and append to this the name of the WAV file we want to play.

It is simple enough to play the wave file directly with this information, but we would be retrieving the data from disk, making our program slower than it needs to be. Instead, we will allocate memory for the sound data using the Windows API `GlobalAlloc` and retrieve a pointer to this memory with the Windows API `GlobalLock`.

In the `OnTimer` event of the joystick component, we call the method `ProcessJoystickInfo`. This returns the current joystick information. Now we call the Windows API `GetCursorPos`, which returns the X and Y coordinates into a structure of type `TPoint`. Now we test the directional information of the joystick, and using the Windows API `SetCursorPos`, we set the new position of the cursor accordingly. Next we test button 1 of the joystick, and if it's pressed we cycle to a new cursor. Finally we check button 2, and if it's pressed, we play the WAV file. Note the flag `SND_ASYNC` in `sndPlaySound`. This flag plays the WAV file asynchronously and the function returns immediately after beginning the sound.

In a nutshell, this completes our joystick component and the demo application. All the source code for the component and example project is on this month's disk. Of course, since this is Delphi, you can easily inherit the component capabilities into your own new components or objects!

Ken Otto and his wife Lorna live in Sacramento, California, USA. Ken is a programmer for Pacific Coast Building Products, writing Pascal applications on the HP3000. Programming in Delphi is a hobby he enjoys. He can be reached at CompuServe 73041,1336