The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Watching TDateTime Values

Q Is there a way to display a `TDateTime` as a date/time string in the debugger and/or watch list? We do a lot of stuff with dates and times and it would be *very* helpful to display a `TDateTime` variable as an actual date/time instead of as a floating point variable. At the moment I have to copy the floating point value from the watch list into a simple application that translates it into a string. It works, but is very tedious.

A If you are using Delphi 5, then the answer is very positive. The debugger now allows you to get the results you want. Compare the Delphi 4 `Add Watch` dialog, shown in Figure 1 with the enhanced dialog from Delphi 5, shown in Figure 2. Notice the primary difference? It's the addition of a checkbox that says `Allow Function Calls`.

The default state for this option is off. But if you turn it on for a watch, you are able to make function calls in the expression being evaluated. This means you can pass your variable to `DateTimeToStr`, `TimeToStr` or `DateToStr`, as required.

The option can be enabled globally, such that the checkbox will be automatically checked for new watches, by selecting `Tools | Debugger Options...`, and on the `General` page of the dialog enabling the `Allow function calls in new watches` checkbox.

Take the small code snippet in Listing 1. If you place a breakpoint on the `ShowMessage` call, you can add some watches which make function calls as mentioned before. This does rely on the functions existing in your application. In order to evaluate these function calls, the debugger calls the routines in your application. If the function is not called by your application (either by your own code, or by VCL code pulled in by the linker), the linker will have stripped it from the application. Figure 3 shows some watches showing a partial degree of success calling functions.

```
var DT: TDateTime;
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  DT := EncodeDate(1999, 5, 26) + EncodeTime(6, 0, 0, 0);
  DT := DT + 365;
  ShowMessage('Date has been increased by one year')
end;
```

➤ *Listing 1*

Executable File Differences

Q I have an archive of a project I built some while ago. Included in the archive are the source files for the project, the options file, resource file (.RES) and also a copy of the final executable. This was built with exactly the same version of 32-bit Delphi as I am currently using (including Update Packs).

I have just tried to rebuild this archived software. Whilst the generated executable is the same size, a file comparison reveals differences. Why could this be?

A This is an obvious source of concern to many software companies. You have a tendency to expect that the same source with the same compiler and linker options will generate exactly the same binary file if fed through the exact same compiler version. Unfortunately, in the Win32 environment, this is often not the case.

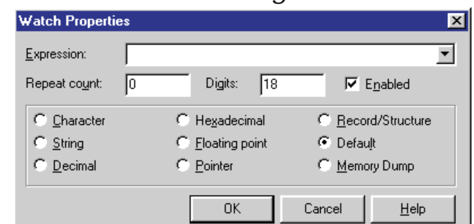
The principal thing here is that the PE file format has various fields reserved for the linker to store date/time stamps in. The main EXE header portion can have a date/time stamp that should indicate when the file image was created. Also the import table, export table, resource tables and so on can also have similar date/time stamps.

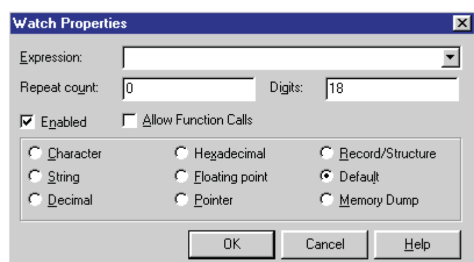
These date/time stamps are `DWord` fields (a `DWord` is the same as a `Cardinal`, an unsigned 32-bit integer) that record the number of seconds elapsed between midnight on January 1st, 1970 and the recorded date/time. C compilers have a

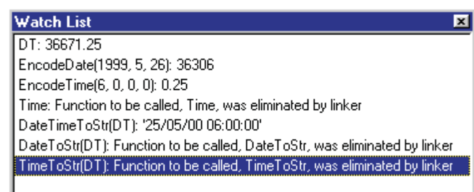
➤ *Below: Figure 1, Adding a watch in Delphi 4.*

➤ *Middle: Figure 2, Adding a watch in Delphi 5.*

➤ *Bottom: Figure 3, Watches calling functions.*







`ctime()` function in their runtime library that can take such a value and display it, but Delphi does not. Consequently, if you wish to view intelligible representations of arbitrary date/time stamps, you need to roll your own version.

However, if you wish to decode the date/time stamps that are present in your executable file, the latest version of TDump does the job. Delphi 4 came with version 5.0.16.4 of TDump. This version listed the time stamp values as hexadecimal numbers and left them at that. Delphi 5 comes with the slightly later version 5.0.16.6. This newer version helps out by calling `ctime()` (it is written in Borland C++) and writing out the represented date and time.

So, to prove the point about these date/time stamps you can compile an application in Delphi once, run TDump and record the details it produces, recompile the project and re-run TDump.

This is what happened when I tested it out. I loaded up a fresh new project into Delphi 5 and pressed `Ctrl+F9` a few times. The reason for the re-compilations was to give Delphi a chance to warm up, but also to avoid another issue which might cloud matters, as will be discussed later.

The generated executable was copied into a directory called One. In a console window I navigated to this directory and generated a TDump log using this command-line:

```
TDump Project1.Exe Dump.Txt
```

I then waited a couple of minutes and re-compiled the same project again. The EXE was copied to a directory called Two and the same command-line was run from there.

Comparing the two dump files revealed that the only difference found by what TDump examined was the resource table time stamp. The project that was compiled first had a timestamp of $27918F0D and the other one had a timestamp of $27919050.

When you compare these findings with what the DOS binary file comparison tool FC.EXE finds, you should see that the date/time stamps are to blame. Listing 2 shows what FC.EXE produced. You can see that the only differences are between values which are parts of the timestamp values.

There is one point about these timestamp values worth bearing in mind. This new version of TDump actually highlights the fact that the Delphi linker uses nonsense values for timestamps. The value of $27918F0D mentioned above decodes to 10:59:25 on Monday January 14th 1991. $27919050 decodes to 11:04:48 on Monday January 14th 1991. As you can see, these are over eight and a half years out of date.

This problem has been reported, but whether it is deemed important enough to fix for Delphi 6 we shall see.

I mentioned earlier that there was another factor which might cloud the issue here. When you make a new project and compile it once, then compare that executable with a second compilation, there will in fact be more differences than just the timestamps. The first compilation will have a different import table and relocation table than all subsequent compilations.

Maybe there is some sorting done by the IDE if the same project is re-compiled, but the outcome is that these two EXE tables are laid out differently. The import table seems to have the same content, just in a different order. I assume that the relocation table also has effectively the same content, but this is more difficult to test.

When archiving an executable, compile it several times first, just to make sure. When doing comparisons, rebuild your comparison project several times too. This will result in fewer file differences.

➤ *Listing 2*

```
Comparing files One\Project1.exe and
Two\project1.exe

00003004: 0D 50
00003005: 8F 90
00003034: 0D 50
00003035: 8F 90
0000304C: 0D 50
0000304D: 8F 90
00003074: 0D 50
00003075: 8F 90
0000308C: 0D 50
0000308D: 8F 90
000030A4: 0D 50
000030A5: 8F 90
000030BC: 0D 50
000030BD: 8F 90
000030D4: 0D 50
000030D5: 8F 90
000030EC: 0D 50
000030ED: 8F 90
00003104: 0D 50
00003105: 8F 90
0000311C: 0D 50
0000311D: 8F 90
```

Common Control Library Version Mismatch

Q I've designed an application where I am using the `TListView` component with the `CheckBoxes` property set to `True`. When I run the application on two machines where Delphi is installed, the checkboxes appear as they should. However, when I install the application on a machine that does not have Delphi installed on it, the checkboxes do not appear. Since the checkboxes don't appear, the users are unable to select the appropriate items with them. Is this a Windows problem, in that a particular DLL is out-of-date or missing?

A This is probably an indication that the machines do not have a recent enough version of ComCtl32.Dll on board. You can download the latest version from Microsoft's website, as I mentioned in Issue 33 (although, regretfully, the precise URL for the file seems to change every now and then...).

According to the MSDN, the checkbox support requires at least version 4.70 of ComCtl32.Dll. This version was distributed with Internet Explorer 3.x. If the target machine has an earlier version, the Microsoft information suggests that you are using a raw installation of Windows 95 or Windows NT 4 with no Internet Explorer.

There have been a number of releases of this DLL (and its associates Shell32.Dll and Shlwapi.dll) with each new one adding new common control features. If you know the required version of the Common Control Library your application needs, you can check that it is installed on the target machine during program initialisation. If an appropriate version is

not installed, you can report an error.

Table 1 shows all the recorded versions of the DLL suite, along with information about what product introduced that version. Also in the table is information about some constants that are defined in the ComCtrls unit in Delphi 4 and later. These constants relate to specific versions of the Common Control Library and can be used to compare against the return value from the function GetComCtlVersion. If this function has already been called, the variable ComCtlVersion from the ComCtrls unit will record this number as well.

Delphi 4 defines the first three constants (ComCtlVersionIE3, ComCtlVersionIE4 and ComCtlVersionIE401) whilst Delphi 5 adds in ComCtlVersionIE5. Presumably Delphi 6 will also define a new ComCtlVersionIE501 constant with the last value from the table, and any others that become necessary.

Since you need version 4.70 for checkbox support in a listview, you check that GetComCtlVersion >= ComCtlVersionIE3 before proceeding normally in your application.

Dynamic Array Question

Q I have a question about dynamic arrays. I can declare a dynamic array of bytes variable and give it a size using a call to SetLength. I wish to initialise the array with a specific value ($FF), so I use a call to FillChar. I'd expect to be able to write the following:

```
FillChar(TheArray,
  Length(TheArray) *
  SizeOf(Byte), $FF)
```

However, after doing this, any access to an array element causes an Access Violation. However, the same approach with a normal array gives no problem. Why?

A This is due to the implementation details of dynamic arrays. Before getting onto that, let's check the parameter details of the FillChar procedure. The declaration of it is shown in Listing 3.

The first parameter is an untyped var parameter; these are passed by reference, which means the *address* of the item passed in is sent across to the routine. Because the parameter is untyped, it does not care what item you pass as the parameter. No type checking is done. Instead, whatever you pass has its address sent through to the procedure.

The implementation of FillChar writes the Value parameter into the memory block that starts at the address passed by the var parameter, for Count bytes.

Now let's check how things work with a normal array. Listing 4 shows code that works just fine with a fixed size array. When the compiler compiles this, it sees that the local variable TheArray requires 100 bytes of storage and ensures that the entry code for this routine (Button2Click in this case) allocates 100 bytes for it on the program stack.

When TheArray is passed to FillChar, the address of the first byte of that 100-byte block is passed through as the first parameter. The address of the beginning of a variable's storage space equates to the address of the variable. FillChar executes its responsibilities by writing the specified value of $FF into that memory location, and also to the 99 memory addresses that sequentially follow it.

To contrast this, let's see what happens with a dynamic array variable. Listing 5 shows code that parallels Listing 4 in the case of a dynamic array, but just as for the questioner, produces an Access Violation because something is not right. When the compiler compiles the dynamic array variable, it does not know how much space the array will ultimately require, but knows that it is a dynamic array. When space is needed for the array (thanks to a call to SetLength) it will be allocated as required, but the program will need to record the address of the allocated memory somewhere.

➤ *Listing 3: The declaration of FillChar.*

```
procedure FillChar(var X; Count: Integer; Value: Byte);
```

➤ *Listing 4: Filling a normal array with a value.*

```
procedure TForm1.Button2Click(Sender: TObject);
var
  TheArray: array[1..100] of Byte;
  Loop: Integer;
begin
  FillChar(TheArray, Length(TheArray) * SizeOf(Byte), $FF);
  for Loop := Low(TheArray) to High(TheArray) do
    ListBox1.Items.Add(Format('Element %.3d has a value of $%x',
      [Loop, TheArray[Loop]]))
end;
```

➤ *Table 1: Versions of the Win32 Common Control Library.*

ComCtl32.DLL Version	Distributed With	ComCtrls Unit Constant	ComCtrls Unit Constant Value
4.00	Windows 95/Windows NT 4		MakeLong(4, 0)
4.70	Internet Explorer 3.x	ComCtlVersionIE3	MakeLong(4, 70)
4.71	Internet Explorer 4.0	ComCtlVersionIE4	MakeLong(4, 71)
4.72	Internet Explorer 4.01 and Windows 98	ComCtlVersionIE401	MakeLong(4, 72)
5.80	Internet Explorer 5	ComCtlVersionIE5	MakeLong(5, 80)
5.81	Windows 2000		MakeLong(5, 81)

Consequently, TheArray is taken to be a pointer. It is a pointer containing the address at which the memory for the dynamic array will reside. The compiler will allocate 4 bytes of memory on the stack for this variable, which is the space required for a pointer (memory addresses are 32 bits wide). You can verify this by evaluating SizeOf(TheArray), which will give 4.

Herein lies the difference between a fixed size array and a dynamic array. As far as writing normal array element access code in Delphi is concerned, they appear exactly the same, using the same syntax. But a dynamic array is really a pointer to the memory space that is dynamically allocated from the heap at runtime, thanks to calls to SetLength. When you access an element of a dynamic array, Delphi de-references the pointer to find where the elements are, then indexes into that memory space to access the array element.

When TheArray is passed to FillChar, the address of this pointer is passed through as the var parameter. FillChar will fill those 4 bytes, along with the next 96 bytes in memory, with $FF. This means that as far as your program is concerned, the dynamic array memory now starts at address $FFFFFFFF. The fact that FillChar fills up the 4-byte dynamic array pointer, along with the following 96 bytes, means that any other local variables declared before the dynamic array variable will be overwritten.

Of course this is very wrong. You need FillChar to write to the memory that represents the dynamic array elements, not the memory that records the address of where those elements are. Consequently, you should try passing either Pointer(TheArray)^ or TheArray[Low(TheArray)] as the first parameter to FillChar.

The Pointer version turns the TheArray variable into what it really is at the machine level, a pointer. It then de-references the pointer to access the array element memory. Because the expression is being passed to a var parameter, FillChar will be passed the address of that array element's address space.

The second option is possibly the better one, as it does not involve any references to pointers. You pass an expression which equates to the first element in the array and the compiler will ensure that that element's address is passed over to FillChar. A benefit of using this approach is that it will work just as well with normal fixed size arrays as well. This allows consistent programming with any type of array.

Either way, the element memory will be overwritten with values, rather than the pointer that holds the element memory address.

Incidentally, before leaving the subject, a similar problem can happen when trying to fill Delphi ShortString variables (which are basically fixed arrays of characters) and 32-bit Delphi long strings. Long strings (which are what we normally use in Delphi these days) are also implemented as a pointer to some dynamically allocated memory.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  TheArray: array of Byte;
  Loop: Integer;
begin
  SetLength(TheArray, 100);
  FillChar(TheArray, Length(TheArray) * SizeOf(Byte), $FF);
  for Loop := Low(TheArray) to High(TheArray) do
    ListBox1.Items.Add(IntToStr(TheArray[Loop]))
end;
```

➤ *Listing 5: Attempting (and failing) to fill a dynamic array with a value.*

TImage.OnProgress Broken?

Q I have tried to use the OnProgress event of a TImage to display a progress bar while my program loads a large (40Mb) bitmap file. Although the Object Inspector will manufacture an event handler for this event, it never gets called (I've set break points everywhere).

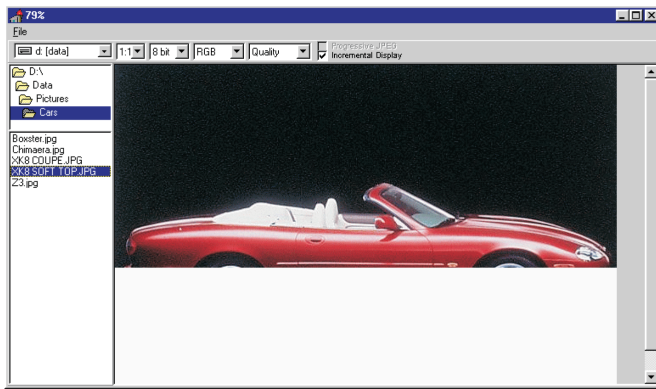
A Three classes define an OnProgress event in the VCL in Delphi 3 and later: TImage, TPicture and TGraphic.

When you set up a TImage OnProgress event handler, the TPicture object that represents the image component's Picture property is told to trigger it as necessary. This is achieved by the image setting the TPicture object's OnProgress event to be handled by a method in the image object.

Similarly, when the picture object has an event handler, it makes sure that whatever graphic object it uses triggers its event where necessary, by assigning one of its own methods to the graphic object's OnProgress event. The buck stops with the graphic object. If it triggers its OnProgress event, this event will be chained up to the TImage component's OnProgress event.

The question is, under what circumstances will a graphic object trigger OnProgress. According to the online help for TGraphic.OnProgress, *"for certain descendants of* TGraphic, OnProgress *occurs during slow processes such as loading, storing, or transforming image data"*. Note that it stipulates this happens with *certain* descendants. The help for the image component's OnProgress event goes on to say: *"Write an* OnProgress *event handler to provide the user with feedback during slow operations such as loading large compressed images."*

Bitmaps are not compressed files. Consequently, TBitmap (as well as TIcon and TMetafile) do not trigger OnProgress. In fact, the only graphic object that comes with Delphi that will trigger OnProgress is TJPEGImage. This class is in the JPEG unit (the source is not installed with the normal VCL source, but you can find it on your Delphi installation CD). Add the unit to any uses clause in your application, and you will be able to load JPEG images, which are then able to trigger the OnProgress event (if they are large, or complex enough).



➤ *Figure 4: Loading a JPEG image with progress indication.*

Figure 4 shows the sample application in Delphi's Help\Examples\JPEG directory, with progress displayed on the caption bar.

Custom Grid Drawing

Q I need to change the colour of a row in a `DBGrid` depending on the value of an item in the underlying table.

A In Delphi 1, you can do custom drawing of a grid cell using the `OnDrawDataCell` event handler. This event is triggered whenever a cell needs to be drawn. If the grid's `DefaultDrawing` property is `False`, then the responsibility for drawing the content of all cells lies with the `OnDrawDataCell` event handler. The event handler is passed four parameters that describe the cell, so that you are well-placed to draw its representation in the grid. The parameters are the grid itself (this is the `Sender` parameter, passed as a `TObject` reference) and a `TRect` record describing which portion of the grid is occupied by the cell. You also get a `TField` reference that represents the field whose value is to be drawn and finally a set parameter describing the state of the cell.

Delphi 2 (and later) also supports this event, but the newer `OnDrawColumnCell` should be used in preference. `OnDrawDataCell` is considered obsolete in these more recent versions and only exists for backward compatibility of source code. The `OnDrawColumnCell` event handler does not get a `TField` reference, but gets a `TColumn` reference instead. `DBGrids` started using column objects in Delphi 2 to represent the attributes of a whole column in the grid. The column object has a reference to the field object that it represents. In addition to the column object, the event handler also gets an integer parameter that indicates the position of the column object in the grid's `Columns` array property.

The `State` set parameter passed to both event handlers is defined to be of type `TGrid- DrawState`, defined as a set of up to three values: `gdSelected`, `gdFocused` and `gdFixed`. `State` will include `gdSelected` if the cell is currently selected, which means that it is the active cell and would normally be filled with the system highlight colour (dark blue by default). It will include `gdFocused` if the cell is currently focused, which means it is the active cell, typically having a dotted line around its border to indicate this. The `State` set will contain `gdFixed` if the cell is a fixed cell, such as a column header cell or a cell in the indicator column.

When the appropriate event handler is called, the grid's canvas object will be set up for the normal drawing. To make minor changes to the cell's appearance, such as changing the background colour, you should be able to simply change the relevant property of the canvas. After this, you call a method of the grid that invokes normal drawing behaviour and it finishes the job for you.

For 16-bit Delphi, some code that does this job is shown in Listing 6. As each cell needs to be drawn, the event handler in the listing is called. This looks at the dataset that the cell's field comes from, and checks to see if the designated criteria are met. In this case, if the TaxRate field (from the `DBDEMOS` Customer.DB table) in the current row is more than zero, then the cell should be drawn with a green background. Since this check will be found true for each cell in the row, the whole row will be drawn green.

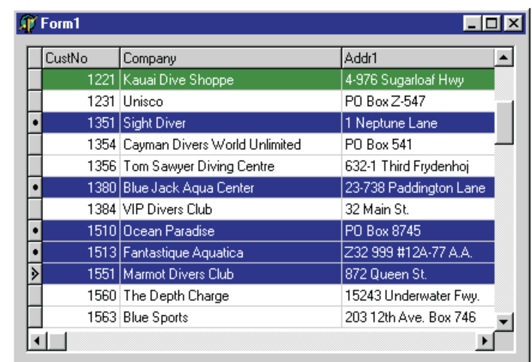
To avoid confusing the display, if the cell is selected, then it is still drawn with the usual blue background. To suggest that it meets the criteria, the font is turned green instead (see Figure 5). If this was not catered for, then the user would have a more difficult task identifying which cells were selected.

After setting the canvas properties, the grid's `DefaultDrawDataCell` method finishes the job off. A small project on this month's disk called Grid16.Dpr contains this code.

In the case of 32-bit Delphi, there is an additional consideration. As well as supporting column objects, `DBGrids` also support multiple selection of entire rows. To enable this, you turn on the `dgMultiSelect` element of the grid's `Options` set property. Once enabled, the user can click on one row to select it and, with the `Ctrl` key held down, click on other rows to select them as well. Alternatively, holding the `Shift` key down allows the up and down cursor keys to select multiple contiguous rows.

Unfortunately, when a row is selected in this way, but is not the 'active' row, the `gdSelected` value does not make it into the `State` set parameter. This means we have to make a specific check to see if the cell being drawn is part of a row that is currently in a multiple selection.

➤ *Figure 5: A grid with multiple selection support and custom cell drawing.*


Form1

CustNo	Company	Addr1
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy
1231	Unisco	PO Box Z-547
1351	Sight Diver	1 Neptune Lane
1354	Cayman Divers World Unlimited	PO Box 541
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj
1380	Blue Jack Aqua Center	23-738 Paddington Lane
1384	VIP Divers Club	32 Main St.
1510	Ocean Paradise	PO Box 8745
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.
1551	Marmot Divers Club	872 Queen St.
1560	The Depth Charge	15243 Underwater Fwy.
1563	Blue Sports	203 12th Ave. Box 746

Multiple selection in a `DBGrid` is implemented by a collection of bookmarks. For each selected record, a bookmark string for that record (as read from the underlying dataset's `Bookmark` property) is added to the `SelectedRows` property (a `TBookmarkList` object). To see if the current row is selected, we must find whether the bookmark string for the current row is in the grid's bookmark list. You can pass a dataset bookmark string to the `IndexOf` method of a `TBookmarkList` object and it will return that bookmark's position. A value of -1 indicates the bookmark is not in the list.

Listing 7 has the appropriate code to take this into account. If the cell is in a multiple selection row, code manually adds the `gdSelected` value into the `State` parameter so the rest of the logic works as before. The method you need to call from an `OnDrawColumnCell` event handler to finish drawing a cell is called `DefaultDrawColumnCell`. You can find this code in Grid32.Dpr on the disk.

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect: TRect;
  Field: TField; State: TGridDrawState);
begin
  with (Sender as TDBGrid) do begin
    { Our criteria means green cells, but if the cell is selected, }
    { stick with the default blue background but use green font }
    if Field.DataSet.FieldByName('TaxRate').AsFloat > 0 then
      if gdSelected in State then
        Canvas.Font.Color := clGreen
      else begin
        Canvas.Brush.Color := clGreen;
        Canvas.Font.Color := clWhite;
      end;
    DefaultDrawDataCell(Rect, Field, State)
  end
```

➤ *Listing 6: 16-bit Delphi code to custom draw certain rows.*

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect: TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
var DataSet: TDataSet;
begin
  with (Sender as TDBGrid) do begin
    DataSet := Column.Field.DataSet;
    //Our criteria means green cells, but if the cell is selected,
    //stick with the default blue background but use green font.
    if DataSet.FieldByName('TaxRate').AsFloat > 0 then begin
      //Selection might be due to multi-selection,
      //which needs separate checking.
      if (dgMultiSelect in Options) and
        (SelectedRows.IndexOf(DataSet.Bookmark) <> -1) then
        Include(State, gdSelected);
      if (gdSelected in State) then
        Canvas.Font.Color := clGreen
      else begin
        Canvas.Brush.Color := clGreen;
        Canvas.Font.Color := clWhite;
      end
    end;
    DefaultDrawColumnCell(Rect, DataCol, Column, State)
  end
```

➤ *Listing 7: 32-bit Delphi code to custom draw certain rows.*