Delphi 2.0

Code Migration Notes

Borland International
2/1/96

Overview

This document describes some of the issues related to migrating 16 bit Delphi 1.0 applications to the new 32 bit version of Delphi 2.0. Although Borland has made every effort to ensure that your code is compatible between versions, there are certain situations that will require code changes in order to move be fully compatible with 32 bit data structures or other changes in the 32 bit Windows operating systems. This document will also provide additional information on optimizing your project for a 32-bit environment and maintaining code which is compatible between both the 16 bit version of Delphi 1.0 and the new 32-bit version of Delphi 2.0.

Working with Strings

In response to customer demand for a more flexible string, Borland has introduced some new string types that support the creation of virtually unlimited size strings. Delphi 2.0 also introduces new character types to fully support localization of applications via the Unicode double byte format. By far the most common issues likely to arise as you migrate to Delphi 2.0 are those dealing with the use and manipulation of Strings.

New String Types

Listed below are the string types supported in Delphi 2.0:

- AnsiString (also referred to as "long string" or "huge string") is the new default string type for Object Pascal. It is comprised of AnsiChar characters and allows for lengths of up to 2 Gigabytes, the operating system limit. It is also compatible with null-terminated strings. This string is always dynamically allocated and automatically garbage collected.
- ShortString is synonymous with the standard String type in Delphi 1.0. It's capacity is limited to 255 characters.
- PAnsiChar is a pointer to a null-terminated AnsiChar string.
  - PWideChar is a pointer to a null-terminated WideChar string for Unicode, double byte strings.
- PChar is a pointer to a null-terminated Char string, which is fully compatible with C-style strings used in Windows API functions. This type hasn't changed from version 1.0 and is currently defined as PAnsiChar.

By default, strings defined in Delphi 2.0 are long strings, that is of the AnsiString type. So if you define a string, as shown below,

```
var
   S: String;    // S is an AnsiString
```

the compiler assumes that you are creating an AnsiString. Alternatively, you can cause variables declared as Strings to instead be of type ShortString using the $H compiler directive. When the value of the $H compiler directive is negative, String variables are ShortStrings, and when the value of the directive is positive (the default), String variables are AnsiStrings. The code below demonstrates this behavior.

```
var
   {$H-}
   S1: String;  // S1 is a ShortString
   {$H+}
   S2: String;  // S2 is an AnsiString
```

The exception to the $H rule is that a String declared with an explicit size (limited to a maximum of 255 characters) is always a ShortString:

```
var
   S: String[63];  // A ShortString of up to 63 characters
```

Note: The $H directive operates on a unit-by-unit basis. Be careful of passing strings declared in units with $H+ to functions and procedures defined in units with $H- and vice-versa.

Setting String Length

In Delphi 1.0 you could set the length of a String by assigning a value to the 0 byte as shown below:

```
S[0] := 23;   { sets the length byte of a short string }
```

This was possible because the maximum length of a short string (255) could be stored in the leading byte. A different physical structure is used to store long strings in the 32 bit version of Delphi 2.0, and therefore the length is stored differently. Therefore, you should call the new SetLength standard procedure to set the length of a string. SetLength is defined as:

```
procedure SetLength(var S: String; NewLength: Integer};
```

SetLength can be used with short or long strings in Delphi 2.0. If you wish to maintain one set of source code for 16-bit Delphi 1.0 projects and 32-bit Delphi 2.0 projects, you can define a SetLength function as follows for 16-bit Delphi 1.0 projects.

```
{$IFDEF WINDOWS}
{ for Delphi 1.0 16 bit projects }
procedure SetLength(var S: String; NewLength: Integer};
begin
   S[0] := Char(NewLength);
end;
{$ENDIF}
```

Dynamically-allocated Strings

In Delphi 1.0 it is possible to use the PString type with the NewStr and DisposeStr standard procedures to implement dynamically-allocated strings. Since long strings are automatically allocated dynamically from the heap, there is no need to use that technique. Change your use of PString to String, and remove the calls to NewStr and DisposeStr.

Indexing Strings as Arrays

Sometimes you want to access a certain character in a string by indexing the string as an array. For example, the following line of code sets the fifth character in the string to ‘A’:

```
S[5] := 'A';
```

This type of operation is still perfectly legitimate with long strings, but there is one caveat. Since long strings are dynamically allocated, you must ensure that the length of the string is greater than or equal to the character element you attempt to index. For example, the following code is invalid:

```
var
   S: String
begin
S[5] := 'A';  // Space for S has not yet been allocated!!
end;
```

whereas the following code is quite valid:

```
var
   S: String
begin
   S := 'Hello';  // allocates enough room for the string
   S[5] := 'A';
end;
```

as is this code:

```
var
   S: String
begin
   SetLength(S, 5); // allocate 5 characters for S
   S[5] := 'A';
end;
```

Null-terminated Strings

When calling Windows 3.1 API functions in Delphi 1.0, programmers had to be aware of the difference between the Pascal String type and the C-style null-terminated strings used in Windows. In Delphi 2.0, the long strings now make it much easier to call Windows API functions. Long strings are both heap-allocated and guaranteed to be null-terminated. For these reasons, you can simply typecast a long string variable when you need to use it as a null-terminated PChar in a Windows API function call. For example, imagine you had a procedure Foo defined as follows:

```
procedure Foo(P: PChar);
```

In Delphi 1.0, you would typically call this function like this:

```
var
  S: String;             { Pascal short string }
  P: PChar;              { null terminated string }
begin
  S := 'Hello world';    { initialize S }
  P := AllocMem(255);    { allocate P }
  StrPCopy(P, S);        { copy S to P }
  Foo(P);                { call Foo with P }
  FreeMem(P, 255);       { dispose P }
```

```
end;
```

Using Delphi 2.0, you could call Foo using a long string variable with the following syntax:

```
var
   S: String;        { a long string is null terminated }
begin
   S := 'Hello World';
   Foo(PChar(S));    { fully compatiple with PChar type}
end;
```

This means that you can optimize your Delphi 2.0 code by removing unnecessary temporary buffers to hold null-terminated strings.

Null-terminated Strings as Buffers

A common use for PChar variables is as a buffer which is passed to an API function which fills the buffer string with information. A classic example of this is the GetWindowsDirectory API function, which is defined in the Win32 API as follows:

```
function GetWindowsDirectory(lpBuffer: PChar; uSize: UINT): UINT;
```

If your goal is to store the Windows directory in a string variable, a common shortcut under Delphi 1.0 is to pass the address of element one of the string as shown below:

```
var
   S: String;
begin
   GetWindowsDirectory(@S[1], 254); { 254 = room for null }
   S[0] := Chr(StrLen(@S[1]));      { adjust length }
end;
```

This technique will not work with Delphi 2.0 long strings for two reasons. First, as mentioned earlier, you must give the string an initial length before any space is allocated. Second, since a long string is already a pointer to heap space, using the @ operator would effectively pass a pointer to a pointer to a character – definitely not what you intended! With long strings, this technique is streamlined by casting the string to a PChar:

```
var
   S: String;
begin
   SetLength(S, MAX_PATH + 1);      { allocate space }
   GetWindowsDirectory(PChar(S), MAX_PATH);
   Set Length(S, StrLen(PChar(S))); { adjust length }
end;
```

PChars as Strings

Since long strings can be used as PChars, it's only fair that the reverse should hold true. Null-terminated strings are assignment-compatible to long strings, so what requires a call to StrPCopy in Delphi 1.0:

```
var
   S: String;
   P: PChar;
begin
   P := StrNew('Object Pascal');
   S := StrPas(P);
   StrDispose(P);
end;
```

can now be accomplished with a simple assignment using long strings:

```
var
   S: String;
   P: PChar;
begin
   P := StrNew('Object Pascal');
   S := P;
   StrDispose(P);
end;
```

Similarly, you can also pass null-terminated strings to functions and procedures expecting String parameters. For example, consider procedure Bar defined as:

```
procedure Bar(S: String);
```

You can call Bar using a PChar as follows:

```
var
   P: PChar;
begin
   P := StrNew('Hello');
   Bar(P);
   StrDispose(P);
end;
```

Delphi 2.0 also offers a standard procedure called SetString which allows you to copy only a portion of a PChar into a String variable. SetString has an advantage in that it works with both long and short strings. The definition of SetString is:

```
procedure SetString(var S: String: Buffer: PChar; Len:
   Integer);
```

New Character Types

Strings, of course, are made up of characters. So there are new character types introduced to support Unicode or wide string types. In addition to the Char type, Delphi 2.0 adds two new character types to the Object Pascal language: AnsiChar and WideChar.

The AnsiChar type is the same as Delphi 1.0's Char type. It is a one-byte character that can contain any one of 256 different characters.

WideChar represents a character which is two-bytes in size. WideChar exists for compatibility with the Unicode character standard adopted by the Win32 API to support localization. Because of it's two-byte size, a WideChar can contain any one of 65,536 possible characters, which is enough for even the largest alphabets.

Currently, the Char and AnsiChar types are one-in-the-same. However, you should never depend on the size of a Char being a certain length in your code – always use SizeOf to determine its actual size.

Variable Sizes

Another issue which may arise as you migrate code to Delphi 2.0 is the fact that some types change size (and therefore range) moving from 16 to 32-bits. The following table illustrates these types:

Type	*16-bit size*	*32-bit size*
Integer	2-byte	4-byte
Cardinal	2-byte	4-byte
String	256-byte	4-byte

For the most part, these new variable sizes will have no effect on your applications. In those areas were you do depend on type sizes, make sure to use the SizeOf() function. Also, if you've written any of these types to binary files or blobs in Delphi 1.0, you will need to take into account the change in size as you read the data back in with Delphi 2.0.

Record Alignment

Also different is the fact that records are aligned on 32-bit boundaries in Delphi 2.0. This is illustrated by the following record:

```
type

TX = record
   B: Byte;
   L: Longint;
end;
```

With the default compiler settings, under Delphi 1.0 SizeOf(TX) returns 5, whereas under Delphi 2.0 SizeOf(TX) returns 8. This is not normally an issue, however, it can be an issue if you don't use SizeOf to determine the size of the record in your code or if you have records written to a binary file.

32-bit Math

A much more subtle issue regarding variable size is that the Delphi 2.0 compiler automatically performs optimized 32-bit math on all operands in an expression, whereas Delphi 1.0 used 16-bit math. Consider the following Object Pascal code:

```
var
   L: longint;
   w1, w2: word;
begin
   w1 := $FFFE;
   w2 := 5;
   L := w1 + w2;
end;
```

Under Delphi 1.0, the value of L at the end of this routine is 3 because the calculation of w1 + w2 is stored as a 16-bit value, and the operation causes the result to wrap. Under Delphi 2.0, the value of L at the end of this routine is $10003 because the w1 + w2 calculation is performed using 32-bit math. The repercussions of the new functionality is that if you use and depend on the compiler's range checking logic to catch errors such as these in Delphi 1.0, then you will need to use some other method for finding those errors in Delphi 2.0 as a range check error will not occur.

The TDateTime Type

In order to maintain compatibility with OLE and the Win32 API, the "zero" value of a TDateTime variable has changed. Date values start at 00/00/0000 under Delphi 1.0 but at 12/30/1899 under Delphi 2.0. This will not effect dates stored in a database field, but it will effect binary dates that you have stored in a binary file or blob.

New Data Types

Delphi 2.0 introduces several new data types including Variant and Currency types. While usage of these types is not required as you migrate your applications to Delphi 2.0, you may wish to take advantage of their power. See the online help or printed documentation for more information.

Other Language Changes

Delphi 2.0 introduces several language changes that you may need to be aware of.

Unit Finalization Section

You can include an optional finalization section in a unit. Finalization is the counterpart of initialization, and takes place when the application shuts down. You can think of the finalization section as "exit code" for a unit. The finalization section corresponds to calls to ExitProc and AddExitProc in Delphi 1.0.

The finalization begins with the reserved word finalization. The finalization section must appear after the initialization section, but before the final end statement.

Once execution enters an initialization section of a unit, the corresponding finalization section is guaranteed to execute when the application shuts down. Finalization sections must handle partially-initialized data properly, just as class destructors must.

Finalization sections execute in the opposite order that units were initialized. For example, if your application initializes units A, B, and C, in that order, it will finalize them in the order C, B, and A.

The outline for a unit therefore looks like this:

```
unit UnitName;
interface
{ uses clause; optional }
...
implementation
{ uses clause; optional }
...
initialization { optional }

...
finalization { optional }

...
end.
```

Assembly Language

Since assembly language is highly dependent on the platform for which it was written, 16-bit built-in assembly language in Delphi 1.0 applications will not work in the new 32-bit Delphi 2.0. You will need to re-write such routines using 32-bit assembly language.

Certain interrupts may or may not be supported under Win32. In some cases, Win32 API functions and procedures take the place of interrupts. If your application makes use of interrupts, you will need to reference the Win32 documentation to see if it is still available.

Additionally, inline hexadecimal code is no longer supported under Delphi 2.0. If you have any routines which use inline code, they should be replaced with 32-bit assembly language routines.

Calling Conventions

Delphi 1.0 has the capability to use either the cdecl or pascal calling convention for parameter passing and stack clean-up for function and procedure calls. The default calling convention for Delphi 1.0 is pascal.

Delphi 2.0 introduces directives which represent two new calling conventions: register and stdcall. The register calling convention is the default for Delphi 2.0, offering faster performance. This method dictates that the first three 32-bit parameters be passed in the eax, edx, and ecx registers respectively. Remaining parameters use the pascal calling convention. The stdcall calling convention is a mixture of pascal and cdecl in that the parameters are passed using cdecl convention but the stack is cleaned-up using the pascal convention.

Note: Whereas functions and procedures in the 16-bit Windows API use the pascal calling convention, Win32 API functions and procedures use the stdcall convention. Consequently, if you have any callback functions in your code, those will also use the stdcall calling convention.

Dynamic Link Libraries (DLLs)

The creation and use of DLLs works very much the same in Delphi 2.0 as 1.0, although there are a few minor differences. Some of these issues are illustrated in the list below:

- Because of Win32's flat memory model, the export directive (which was necessary for callback and DLL functions in Delphi 1.0) is not necessary under Delphi 2.0. It is simply ignored by the compiler.
- The preferred way to export functions in a Win32 DLL is by name (as opposed to by ordinal). To optimize loading of your functions, use the resident directive with each function or procedure in the exports clause.
- Exported names are case-sensitive. Therefore, you must use proper case when importing functions by name and when calling GetProcAddress.
- When you import a function or procedure and specify the library name after the external directive, the file extension can be included. If no extension is specified, ".dll" is assumed.
- Under Windows 3.x, a DLL in memory only has one data segment which is shared by all instances of the DLL. Therefore, if applications A and B both load DLL C, changes made to global variables in DLL C from application A would be visible to application B, and vice-versa. Under Win32, each DLLs receives its own data segment, so changes made to global DLL data from one program are not visible from another program.

Third-Party Components

Delphi 2.0 compiled units (.dcu files) differ from those of Delphi 1.0. For this reason, if your Delphi 1.0 projects make use of any third-party components, you will need to recompile the source code for the units containing those components before they can be used in Delphi 2.0. If you do not have the source code to a particular third party component, you should contact your vendor for a 32-bit version of the component.

Windows Operating System Changes

There are several areas where changes in the 32 bit architecture of Windows can impact code written in Delphi. These include changes resulting from the 32-bit memory model, changes in resource formats, unsupported features and changes to the Windows API itself.

32-Bit Address Space

Win32 provides a 4 Gigabyte flat address space for your application. What "flat" means is that all segment registers hold the same value, and the definition of a pointer is an offset into that 4 GB space. Because of this, any code in your Delphi 1.0 applications which depend on the concept of a pointer consisting of selector and offset must be rewritten to accommodate the new architecture.

The following elements of the Delphi 1.0 runtime library are 16-bit pointer-specific, and are not in the Delphi 2.0 runtime library: DSeg, SSeg, CSeg, Seg, Ofs, and SPtr.

Because of the way Win32 uses a hard disk paging file to simulate RAM-on-demand, the Delphi 1.0 MemAvail and MaxAvail functions are no longer useful for gauging available memory. If you need to obtain this information in Delphi 2.0, you should use the GetHeapStatus Win32 API function which is defined as follows:

```
function GetHeapStatus: THeapStatus;
```

The THeapStatus record is designed to provide information (in bytes) on the status of the heap for your process. This record is defined as:

```
type
  THeapStatus = record
    TotalAddrSpace: Cardinal;
    TotalUncommitted: Cardinal;
    TotalCommitted: Cardinal;
    TotalAllocated: Cardinal;
    TotalFree: Cardinal;
    FreeSmall: Cardinal;
    FreeBig: Cardinal;
    Unused: Cardinal;
    Overhead: Cardinal;
    HeapErrorCode: Cardinal;
  end;
```

Again, since the nature of Win32 is such that the amount of "free" memory has little meaning, most users will find the TotalAllocated field (which indicates how much heap memory has been allocated by the current process) most useful for debugging purposes.

32-bit Resources

If you have any resources (.res or .dcr file) which you link into your application or use with a component, you will have to create 32-bit versions of these files before they can be used with Delphi 2.0. Typically, this is a simple matter of using the included Image Editor or a separate resource editor such as Resource Workshop to save the resource file in a 32-bit compatible format.

VBX controls

Microsoft does not support the older 16-bit VBX controls in 32-bit applications on Windows 95 and Windows NT, so they are therefore not supported in Delphi 2.0. OLE controls (OCXs) effectively replace VBX controls on 32-bit platforms. If you need to migrate a Delphi 1.0 application that uses VBX controls, you should contact your VBX vendor to get an equivalent 32-bit OCX control.

Changes to the Windows API functions

Some Windows APIs or features have changed from Windows 3.1 to Win32. Some 16-bit API functions no longer exist in Win32, some functions are obsolete but continue to exist for compatibility's sake, and some accept different parameters or return different types or values. The following sections provide you with a list of these such functions. For complete information, refer to the Win32 documentation.

Obsolete Windows 3.x API functions

Win 3.x Function	*Win32 replacement*
OpenComm	CreateFile
CloseComm	CloseHandle
FlushComm	PurgeComm
GetCommError	ClearCommError

ReadComm	ReadFile
WriteComm	WriteFile
UngetCommChar	N/A
DlgDirSelect	DlgDirSelectEx
DlgDirSelectComboBox	DlgDirSelectComboBoxEx
GetBitmapDimension	GetBitmapDimensionEx
SetBitmapDimension	SetBitmapDimensionEx
GetBrushOrg	GetBrushOrgEx
GetAspectRatioFilter	GetAspectRatioFilterEx
GetTextExtent	GetTextExtentPoint
GetViewportExt	GetViewportExtEx
GetViewportOrg	GetViewportOrgEx
GetWindowExt	GetWindowExtEx
GetWindowOrg	GetWindowOrgEx
OffsetViewportOrg	OffsetViewportOrgEx
OffsetWindowOrg	OffsetWindowOrgEx
ScaleViewportExt	ScaleViewportExtEx
ScaleWindowExt	ScaleWindowExtEx
SetViewportExt	SetViewportExtEx
SetViewportOrg	SetViewportOrgEx
SetWindowExt	SetWindowExtEx
SetWindowOrg	SetWindowOrgEx
GetMetafileBits	GetMetafileBitsEx
SetMetafileBits	SetMetafileBitsEx
GetCurrentPosition	GetCurrentPositionEx
MoveTo	MoveToEx
DeviceCapabilities	DeviceCapabilitiesEx
DeviceMode	DeviceModeEx
ExtDeviceMode	ExtDeviceModeEx
FreeSelector	N/A
AllocSelector	N/A
ChangeSelector	N/A
GetCodeInfo	N/A
GetCurrentPDB	GetCommandLine and/or GetEnvironmentStrings
GlobalDOSAlloc	N/A
GlobalDOSFree	N/A
SwitchStackBack	N/A
SwitchStackTo	N/A
GetEnvironment	(Win32 file I/O functions)
SetEnvironment	(Win32 file I/O functions)
ValidateCodeSegments	N/A
ValidateFreeSpaces	N/A
GetInstanceData	N/A
GetKBCodePage	N/A
GetModuleUsage	N/A
Yield	WaitMessage and/or Sleep
AccessResource	N/A
AllocResource	N/A
SetResourceHandler	N/A

AllocDSToCSAlias	N/A
GetCodeHandle	N/A
LockData	N/A
UnlockData	N/A
GlobalNotify	N/A
GlobalPageLock	VirtualLock
CreatePQ	N/A
MinPQ	N/A
ExtractPQ	N/A
InsertPQ	N/A
SizePQ	N/A
DeletePQ	N/A
OpenJob	N/A
StartSpoolPage	N/A
EndSpoolPage	N/A
WriteSpool	N/A
CloseJob	N/A
WriteDialog	N/A
DeleteSpoolPage	N/A

Win32 API compatibility functions

Win 3.x Function	*Win32 replacement*
DefineHandleTable	N/A
MakeProcInstance	N/A
FreeProcInstance	N/A
GetFreeSpace	GlobalMemoryStatus
GlobalCompact	N/A
GlobalFix	N/A
GlobalUnfix	N/A
GlobalWire	N/A
GlobalUnwire	N/A
LocalCompact	N/A
LocalShrink	N/A
LockSegment	N/A
UnlockSegment	N/A
SetSwapAreaSize	N/A

16 and 32-bit Concurrent Projects

This section is intended to give you some guidelines as to developing project which will compile under 16-bit Delphi 1.0 or the new 32-bit version of Delphi 2.0. While you can follow the directions outlined in this paper for source-code compatibility, here are some further pointers to help you along:

- The WINDOWS conditional is defined by the compiler under Delphi 1.0, while the WIN32 conditional is defined under Delphi 2.0. You can use these defines to perform conditional compilation with the {$IFDEF WINDOWS} or {$IFDEF WIN32} directives.
- Avoid the use of any component or feature in Delphi 2.0 not supported under Windows 3.1 or Delphi 1.0 if you want to recompile with Delphi 1.0 for a 16-bit application. For example, you should avoide the use of the Win95 components and features such as multi-

threading which are not available on Windows 3.1. The easiest way to ensure compatibility with projects between Delphi 1.0 and Delphi 2.0 is to develop in Delphi 1.0 and recompile with Delphi 2.0 for optimized 32-bit performance.

- Be wary of differences between the APIs. If you need to use an API procedure or function which is implemented differently on the different platforms, make use of the WINDOWS and WIN32 conditional defines.
- Version 2.0 components often have more or different properties than their 1.0 versions. When version 2.0 components are saved out to a .dfm file, these properties are written as well. While it's often possible to "ignore" the errors that occur when loading projects with these properties under Delphi 1.0, it's often a more favorable solution to maintain two separate sets of .dfm files for each platform.