Creating A Delphi Build Process

by Dave Collie

If you work in a team collaborating on a single product then you should be performing regular builds of the system. A build creates all of the product's target files ready for release and is essentially an automated compile for the entire application, including all executables, DLLs and resource files etc. It's important when working in a team to perform builds on a regular basis so that you can guarantee the code has not been broken somewhere along the line.

In a normal team development scenario where some form of source code management system is in use, for example Microsoft Visual Source Safe, each developer normally has their own 'sandbox' of code. A sandbox is a set of code unique to the developer which can be worked on in isolation to the rest of the team. This enables the developer to make changes to the code without affecting the other team members. The downside of this is that the different sandboxes can get out of sync with each other and when all the team members' code has been checked in, changes made by different developers can be incompatible, resulting in a system that no longer compiles. The purpose of a build is to attempt to compile the system in its entirety and to ensure that this has not happened.

As I previously indicated, builds should be done on a regular basis, anywhere from every day to once a week. In fact the frequency of the builds should be inversely proportional to the size of the development team, because the more team members there are, the more chance the code will get out of line. Ideally, a build should include all end-user deliverables, not only code but also help and so on. It's well documented that Microsoft, as a matter of course, perform daily builds of a lot of their products. Microsoft have taken the build process to an extreme and team members who 'break' the build can be subject to humiliation. I don't recommend this approach myself, but it may appeal to you... Working in smaller teams of say two to three people you can afford to relax the build to once a week because there's normally better communication within the team.

Builds can also be useful management tools for highlighting the state of development. If the product regularly fails the build then the chances are that it's in a mess and needs sorting out. It also provides management with visible progress: if each build adds new functionality you know you're moving forward. Without a successful build to show progress it's normally a case of guesswork as to what state the product is in.

The most important feature of a build process is that it must be *repeatable*. That is, if the build is done twice in a row you will get the same results. This is important because without the process being repeatable you won't know whether the build process or the product lies at fault when a problem in the build occurs. This rules out doing each step of the build process manually: it must be automated. If you have a product that consists of multiple Delphi projects it's not good enough to just load each project one by one into the IDE and perform manual compilations. This is too open to errors: it's easy to imagine a developer's IDE compiler settings might have changed from one build to the next. Also, what if the executables produced by the build require certain compiler flags to be used that are different to those required for day-to-day development (for example no assertions or debug symbols)? In this case if the build were done manually the developer would have to change each project's options by hand, which is obviously error prone.

A Delphi Build Process

Luckily it's quite easy to create an automated build process using Delphi's command line compiler, DCC32. This is a console application that can compile any Delphi project: executables, DLLs and, in Delphi 3 and 4, packages. By creating a DOS batch file that calls DCC32 for each project in the product, an automated repeatable build process can quickly be assembled. If you use a source code management system which provides command line tools (most do) then the batch file can also get the latest version of your code before compiling it. This is much better than getting the code by hand, since manual processes are error prone. Although primitive by today's standards, a batch file is useful for automating the build process as it is easy to write and can normally perform all of the tasks necessary for automation.

Before creating an example DOS batch build process, it's important to understand DCC32 fairly well. It has a large number of command line options, which enables it to match all of the compilation configurations that can be defined in the Delphi IDE. Table 1 lists all of the options and explains each one briefly (all of the options are explained in detail in Appendix A of the *Delphi User's Guide*). Each option can be preceded on the command line with a / or -.

It would be tedious to have to define each of these options for every project where the options are product wide (ie apply to all projects). However, you can set up a list of common options in a configuration file that is processed by

➤ *Listing 1*

```
-aWinTypes=Windows;WinProcs=Windows;DbiProcs=BDE;DbiTypes=BDE;DbiErrs=BDE
-u"d:\Program Files\Borland\Delphi 3\Lib"
```

DCC32 before compiling a project. This file is called DCC32.CFG and can reside either in the current directory or the same directory as DCC32.EXE. As shipped, Delphi provides a default configuration file in its \Bin subdirectory. Listing 1 shows the default settings. As you can see, it defines the default unit aliases and library path.

The last thing to be aware of before creating a batch file to call DCC32.EXE is that if a project source references any units using relative paths, as opposed to absolute paths, then the current directory must be set to be the directory where the project is located before calling the compiler. If this is not done then DCC32 will be unable to locate the units since it looks for them relative to the current directory. It is therefore good practice to always change directory to the project directory before launching the compiler.

Now that I've covered the fundamentals of using DCC32, we can now create a batch file to perform the build process. I've included some example programs on the disk that for the purpose of this article represent our product, along with the batch files I'll be creating. Before running any of the batch files make sure you're in the \Build subdirectory.

Listing 2 shows the code for the first, most basic, build process. It compiles both of the projects in the product in non-verbose (quiet) mode and outputs a message depending on the success or failure of the build. This is done using the rudimentary flow control available in batch files. If DCC32 fails to compile a project, it returns an exit code of 1. This can then be tested by looking at the `ERRORLEVEL` variable and taking action depending on its value. In this case we use `GOTO` to abort the build if any compilation fails.

The next build process places all executable files into a known directory, in this case the \BuildOutput sub-directory, as shown in Listing 3. The output directory is specified using the `/E` compiler option. It would be very easy to make this a parameter to the batch file by changing `/E..\BuildOutput` to `/E%1`. The output directory would then be specified as the first parameter to the batch file, for example `OutputBuild C:\Temp` would place all the executable files in the C:\Temp directory.

The basic build process can now be expanded to cope with more demands. Typically more than one type of build is required, say debug and production builds. Debug builds are given to QA for testing and contain extra code to help catch problems but are larger and slower than production builds due to the extra 'defensive' code. Production builds are meant for the customer and are tuned to be lean and mean. Creating a batch file for a debug build requires the use of something not yet discussed: compiler directives. A compiler directive allows control over the

Option	IDE Equivalent (Delphi 3, in italics) / Comment
/A	*Environment Options|Library|Unit Aliases*. Sets an alias for a unit, eg /AWinTypes=Windows;WinProcs=Windows
/B	*Project|Build All*. Build all units.
/CC	*Project Options|Linker|Generate console application.* Generates a console (non-GUI) application. Applicable to executables only.
/CG	*Project Options|Linker|Generate console app (unchecked).* Generates a GUI application.
/D	*Project Options|Directories/Conditionals|Conditional defines.* Define conditional symbol(s). Used to define any conditional symbols the project requires, eg /DMULTIUSER;SQL defines both the MULTIUSER and SQL symbols.
/E	*Project Options|Directories/Conditionals|Output directory.* Target directory to place executables and libraries. If a map file is generated then it is also created in this directory, eg /E"C:\My Project\Latest Build"
/F	*Search|Find Error*. Find location of run-time error.
/GS	*Project Options|Linker|Map File|Segments.* Create a map file with segment information only.
/GP	*Project Options|Linker|Map File|Publics.* Create map file with publics information only.
/GD	*Project Options|Linker|Map File|Details.* Create detailed (full) map file.
/H	*Project Options|Compiler|Show Hints.* Output hint messages.
/I	*Project Options|Directories/Conditionals|Search Path.* Include file directories. Specifies the directory for the compiler to search for files included with the $I directive if the files cannot be found in the normal search path, eg /I"C:\My Project\Include Files";"D:\Standard Include Files"
/J	*Project Options|Linker|Generate Object Files* Generates standard binary object (OBJ) files instead of DCUs. In a multiple language development it is possible to link these object files into C programs.
/JP	*N/a.* Generates binary object files that are compatible with C++.
/K	*Project Options|Linker|Image base*. Sets the preferred load address of the compiled image (normally only important for libraries).
/LE	*Environment Options|Library|DPL output directory.* Defines the directory to place compiled packages. Equivalent to /E for executables.
/LN	*Environment Options|Library|DCP output directory.* Defines the directory to place compiled package DCP files. Equivalent to /E for executables.
/LU	*N/a.* Defines any run-time packages the application requires *in addition* to those defined in the IDE's project options dialog.
/M	*Project|Compile*. Compiles units only if necessary, eg if the units are out of date.
/N	*Project Options|Directories/Conditionals|Unit output directory.* Defines the directory to place DCU files. Equivalent to /E for executables.
/O	*Project Options|Directories/Conditionals|Search Path.* Object file directories. Specifies the directory to search for object files linked in with the $L directive.
/P	*N/a.* Instructs the compiler to look for 8.3 file names as well as long file names.
/Q	*N/a.* Quiet compilation. If this option is not turned on then the compiler outputs a line for every unit it compiles to the standard output.
/R	*Project Options|Directories/Conditionals|Search Path.* Resource file directory. Specifies the directory to search for resource files (.RES) included with the $R directive.
/TX	*Project Options|Application|Target file extension*. Defines the extension of the target file, eg /TXOCX would generate a file with a .OCX extension.
/U	*Environment Options|Library|Library Path.* Defines the directory to search for units that are not explicitly referenced in the project's source. This is a very important option that must reflect your normal library path.
/V	*Project Options|Linker|Include TD32 debug info.* Instructs the compiler to generate Turbo Debugger 5 compatible debugger inforation at the end of the executable file.
/W	*Project Options|Compiler|Show Hints.* Output warning messages.
/Z	*Package Options|Description|Explicit rebuild*. This instructs the compiler to prevent implicit compilation at a later date. This applies to both packages and units.

➤ *Table 1: DCC32 options.*

```
@ECHO OFF
REM BasicBuild.bat
REM This just compiles the product in using default compiler options.
REM Build the first project.
CD "..\My Project 1"
DCC32 /Q MyProject1.dpr
IF ERRORLEVEL 1 GOTO FAILED
REM Now build the second project.
CD "..\My Project 2"
DCC32 /Q MyProject2.dpr
IF ERRORLEVEL 1 GOTO FAILED
ECHO My Product built OK
GOTO END
:FAILED
ECHO My Product failed to build
:END
CD ..\Build
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
@ECHO OFF
REM OutputBuild.bat
REM This compiles the product using default compiler options
REM and places the executables in a named directory.
REM Build the first project.
CD "..\My Project 1"
DCC32 /Q /E..\BuildOutput MyProject1.dpr
IF ERRORLEVEL 1 GOTO FAILED
REM Now build the second project.
CD "..\My Project 2"
DCC32 /Q /E..\BuildOutput MyProject2.dpr
IF ERRORLEVEL 1 GOTO FAILED
ECHO My Product built OK
GOTO END
:FAILED
ECHO My Product failed to build
:END
CD ..\Build
```

```
@ECHO OFF
REM DebugBuild.bat
REM This compiles the product in a debug state and places the executables in a
named directory.
REM Build the first project.
CD "..\My Project 1"
DCC32 /B /H /W /Q /$D+ /$C+ /$O- /E..\BuildOutput MyProject1.dpr
IF ERRORLEVEL 1 GOTO FAILED
REM Now build the second project.
CD "..\My Project 2"
DCC32 /B /H /W /Q /$D+ /$C+ /$O- /E..\BuildOutput MyProject2.dpr
IF ERRORLEVEL 1 GOTO FAILED
ECHO My Product built OK
GOTO END
:FAILED
ECHO My Product failed to build
:END
CD ..\Build
```

➤ *Listing 5*

features of the compiler itself, such as whether or not to compile with debug information. Directives can be embedded into source code directly: Listing 2 shows the use of compiler directives in code to turn debug information and assertions on and optimisation off, a normal debug build state. Directives can either be short or long, eg `$D+` or `$DEBUGINFO ON` both direct the compiler to turn debug information on. It is not ideal to embed these types of directives into the source code because they would have to be changed every time a different type of build (production or debug) was performed. The directives in Listing 4 have IDE equivalents in the `Compiler` page of the `Project Options` dialog and so are normally set from there.

It is possible to define directives when calling the command line compiler, using the `/$` option. This option is followed by the short directive identifier (DCC32 does not recognise long directive identifiers). When specifying a directive on the command line it must be followed by + to turn it on or - to turn it off: eg `/$D+` turns debug information on.

Listing 5 is a batch file that uses some compiler directives to create a debug build by turning debug information and assertions on and optimisation off. It also ensures that all units are built for every project. Although the `/M` (make) option compiles quicker it's important to make sure that the build is

```
Unit MyUnit;
{$DEBUGINFO ON}
{$ASSERTIONS ON}
{$OPTIMIZATION OFF}
...
```

➤ *Listing 4*

complete and that nothing missed compilation. The make option suffers from the same problem as the IDE's `Compile` option, in that units are not recompiled if a compiler option/directive has changed but the code has remained the same. The `/H` and `/W` options are included to output hints and warnings that may occur. If the build does produce hints or warnings then strictly speaking it should be considered 'broken' since not all of the projects compiled cleanly.

As an aside, it is good practice to create a `boolean` constant that is defined to be true if debug information is on and false if it is not. This constant can then be tested and debug code only executed if the constant is true. Listing 6 shows a code snippet from the `Debug` unit that defines a debug constant and Listing 7 shows some code from `MyProject1\Main.pas` that uses it. If debug information is turned off and optimisations turned on, the debug code will be factored out by the compiler since the constant will be false and the compiler knows it will never be executed.

Sundry Files

As well as Delphi projects, there are other file types that may be included in a build, as I mentioned in the introduction. If you use resource files then these will need to be compiled as well, before any projects that use them are compiled. Using the Borland resource compiler (brcc32.exe) to compile a resource script creates a binary resource file with the extension of .RES. This resource file is then bound into a project by using the `$R` directive in the project's source unit. MyProject1.dpr shows an example of this: it uses a user defined resource to bind the contents of a default untitled file into the executable. This resource is then saved to a file when the program is run (see Listing 8) and

loaded into the editing window when the user chooses to create a new file (see Listing 9).

Listing 10 shows a batch file that compiles the resource script (ExtraData.rc) to generate the binary resource file bound into the MyProject1 executable. It also compiles the help file for the product using the Help Workshop program, HCW.EXE. This is supplied with Delphi (in the Help\Tools subdirectory) and is used for creating and compiling help project files. The command line switches used compile the help project (/C) and then exit (/E).

Conclusion

It's a small step from the DebugBuild batch file to a batch file that performs a production build. Normally this entails turning off all the debug related features of the compiler and making sure all optimisations are turned on. If you find batch files too limiting then you can consider other alternatives such as writing a Delphi application that performs the build, or a Windows batch language shell program *[We use BATSH.EXE, see www.fmi.ch/groups/ThomasNyffenegger/Group.html. Ed]*. Or you might consider using the emerging Windows scripting technology, Windows Scripting Host: it is currently immature but shows promise by allowing you to write VBScript or JavaScript applets that can be executed in either console or GUI mode. This article should have given you the grounding to follow any of these options. However, if you decide to implement a build process you will find that the initial investment in developing it will be paid back later with better production code.

Dave Collie is a senior Delphi and OO design consultant with Informatica Consultancy & Development, specialising in the design and implementation of large applications in a fully object oriented environment. He can be contacted via email at dave@informatica.uk.com

```
const
  // Define constant DebugOn to be true if debug info is on.
  DebugOn = {$IFOPT D+} True {$ELSE} False {$ENDIF};
  // Define constant AssertionOn to be true if assertions are on.
  AssertionsOn = {$IFOPT C+} True {$ELSE} False {$ENDIF};
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
procedure TMainForm.FileSend(Sender: TObject);
var
  MapiMessage: TMapiMessage;
  MError: Cardinal;
  MessageText: String;
begin
  // This will be compiled out if debug information is turned off.
  if DebugOn then begin
    RichEdit1.Lines.Add('Message created on ' +
      FormatDateTime('hh:mm:ss ddd mmm yyyy', Now));
  end;
  MessageText := RichEdit1.Lines.Text;
  with MapiMessage do begin
    ulReserved := 0;
    lpszSubject := nil;
    lpszNoteText := PChar(MessageText);
    lpszMessageType := nil;
    lpszDateReceived := nil;
    lpszConversationID := nil;
    flFlags := 0;
    lpOriginator := nil;
    nRecipCount := 0;
    lpRecips := nil;
    nFileCount := 0;
    lpFiles := nil;
  end;
  MError := MapiSendMail(0, 0, MapiMessage, MAPI_DIALOG or MAPI_LOGON_UI
    or MAPI_NEW_SESSION, 0);
  if MError <> 0 then MessageDlg(rsSendError, mtError, [mbOK], 0);
end;
```

```
procedure TMainForm.FormCreate(Sender: TObject);
var ResourceData: TResourceStream;
begin
  Application.OnHint := ShowHint;
  // Extract default untitled file.
  ResourceData :=
    TResourceStream.Create(HInstance, 'DefaultUntitled', RT_RCDATA);
  try
    ResourceData.SaveToFile(ExtractFilePath(Application.ExeName)+'Untitled.txt');
  finally
    ResourceData.Free;
  end;
  FileNew (Self);
end;
```

➤ *Above: Listing 8*

➤ *Below: Listing 9*

```
procedure TMainForm.FileNew(Sender: TObject);
begin
  FFileName := rsUntitled;
  RichEdit1.Lines.LoadFromFile(ExtractFilePath(Application.ExeName)+
    'Untitled.txt');
  RichEdit1.Modified := False;
end;
```

```
@ECHO OFF
REM FullBuild.bat
REM This compiles the product in a debug state and places
REM the executables and sundry files into a named directory.
REM Build the first project.
CD "..\My Project 1"
BRCC32 ExtraData.rc
DCC32 /B /H /W /Q /$D+ /$C+ /$O- /E..\BuildOutput MyProject1.dpr
IF ERRORLEVEL 1 GOTO FAILED
REM Now build the second project.
CD "..\My Project 2"
DCC32 /B /H /W /Q /$D+ /$C+ /$O- /E..\BuildOutput MyProject2.dpr
IF ERRORLEVEL 1 GOTO FAILED
REM Make the help file.
CD ..\HELP
HCW /C /E myproduct.hpj
IF ERRORLEVEL 1 GOTO FAILED
COPY MyProduct.hlp ..\BuildOutput
ECHO My Product built OK
GOTO END
:FAILED
ECHO My Product failed to build
:END
CD ..\Build
```

➤ *Listing 10*