Borland C++Builder tools overview

Borland C++ includes many tools to help you create C++ programs.

The following table lists the Borland tools that come with your Borland C++ package:

File	Description
BCC32.EXE	C++ compiler (32-bit), command-line version
BRC32.EXE	Resource compiler (32-bit), command-line version
BRCC32.EXE	Resource shell (32-bit)
CPP32.EXE	C preprocessor (32-bit), command-line version
DCC32.EXE	Object Pascal compiler (32-bit), command-line version
GUIDGEN.EXE	Generates unique Globally Unique IDs for use in OLE apps
GREP.EXE	File search utility
IDETOMAK.EXE	Converts Borland C++ .IDE project files into C++Builder project .MAK files
ILINK32.EXE	Incremental linker (32-bit), command-line version
IMPDEF.EXE	Utility used when building apps with Libraries
IMPLIB.EXE	Utility used when building apps with Libraries
JITIME.EXE	Utility used to register just-in-time debugger
MAKE.EXE	Make utility
TDUMP.EXE	File dump utility
TLIB.EXE	Utility for maintaining static-link libraries
TLINK32.EXE	Linker (32-bit), command-line version
TOUCH.EXE	Change files stamps to current date/time
TRIGRAPH.EXE	Converts 3-character trigraph sequences into single characters
WS32.EXE	WinSight32, 32-bit utility to watch window messages

Using the C++ command-line compiler

See also

C++Builder includes BCC32.EXE, the Borland C++ command-line compiler.

Command-line compiler syntax

The syntax for BCC32 is:

```
BCC32 [option [option...]] filename [filename...]
```

Where:

- Items enclosed in brackets are optional.
- `option` refers to the command-line options.
- `filename` refers to the source-code files you want to compile.

To see a list of the commonly used compiler options, type `BCC32` at the command line (without any options or file names), then press Enter. This list displays the options that are enabled by default.

The command-line compiler name, each option, and the file names must all be separated by at least one space. Precede each option by either a hyphen (-) or a forward slash (/). For example:

```
BCC32 -Ic:\code\hfiles
```

Options and file names entered on the command line override settings in configuration files.

You can use BCC32 to send .OBJ files to TLINK32 or .ASM files to TASM32 (if you have TASM32 installed on your machine).

Default settings

BCC32.EXE has specific options that are on by default. To turn off a default option or to override options in a configuration file, follow the option with a minus (-) sign.

Files without extensions and files with the .CPP extension compile as C++ files. Files with a .C extension or with extensions other than .CPP, .OBJ, .LIB, or .ASM compile as C files.

The compilers try to link with a module-definition file with the same name as the executable. To link with a module-definition file with a different name, specify the name on the compiler command line. You must include the .DEF extension and you cannot link with more than one module-definition file.

Using compiler configuration files

See also

If you repeatedly use a certain set of options, you can list them in a configuration file instead of continually typing them on the command line. A configuration file is a standard ASCII text file that contains one or more command-line options. Each option must be separated by a space or a new line.

Whenever you issue a compile command, BCC32.EXE searches for a configuration file called BCC32.CFG. The compiler looks for the .CFG file first in the directory where you issue the compile command, then in the directory where the compiler is located.

You can create and use multiple configuration files in addition to using the default .CFG file. To use a configuration file, use the following syntax where you would place the compiler options:

```
+[path]filename
```

For example, you could use the following command line to use a configuration file called MYCONFIG.CFG:

```
BCC32 +C:\MYPROJ\MYCONFIG.CFG mycode.cpp
```

Options typed on the command line override settings stored in configuration files.

Using compiler response files

See also

Response files let you list both compiler options and file names in a single file (unlike configuration files, which accept only compiler options). A response file is a standard ASCII text file that contains one or more command-line options and/or file names, with each entry in the file separated by a space or a new line. In addition to simplifying your compile commands, response files let you issue a longer command line than most operating systems allow.

The syntax for using a single response file is:

```
BCC32 @[path]respfile.txt
```

The syntax for using multiple response files is:

```
BCC32 @[path]respfile.txt @[path]otheresp.txt
```

Response files shipped with C++Builder have an .RSP extension.

Options typed at the command line override any option or file name in a response file.

Compiler-option precedence rules

See also

The command-line compilers evaluate options from left to right, and follows these rules:

- If you duplicate any option (except for the options **-D**, **-U**, **-I**, or **-L**), the last option typed overrides any earlier one. (**-D**, **-U**, **-I**, and **-L** are cumulative.)
- Options typed at the command line override configuration and response file options.

Entering directories for command-line options

See also

C++Builder can search multiple directories for include and library files. This means that the syntax for the library directories (**-L**) and include directories (**-I**) command-line options, like that of the **#define** option (**-D**), allows multiple listings of a given option. Here is the syntax for these options:

```
-Ldirname[;dirname;...]
-Idirname[;dirname;...]
```

The parameter `dirname` used with **-L** and **-I** can be any directory or directory path. You can enter these multiple directories on the command line in the following ways:

- You can stack multiple entries with a single **-L** or **-I** option by using a semicolon:
  ```
  BCC32.EXE -Ldirname1;dirname2;dirname3 -Iinc1;inc2;inc3 myfile.c
  ```
- You can place more than one of each option on the command line, like this:
  ```
  BCC32.EXE -Ldirname1 -Ldirname2 -Ldirname3 -Iinc1 -Iinc2 -Iinc3 myfile.c
  ```
- You can mix listings:
  ```
  BCC32.EXE -Ldirname1;dirname2 -Ldirname3 -Iinc1;inc2 -Iinc3 myfile.c
  ```

If you list multiple **-L** or **-I** options on the command line, the result is cumulative: The compiler searches all the directories listed, in order from left to right.

Note: The C++Builder environmant also supports multiple library directories.

CPP and CPP32: The Preprocessors

See also

CPP and CPP32 each produce a list (in a file) of a C program in which **#include** files and **#define** macros have been expanded. While you do not need to use the preprocessors during normal compilation, you may find the list file helpful for debugging purposes.

Note: CPP (16-bit) and CPP32 (32-bit) both produce the same type of list file. For simplification, this document uses the term CPP to represent both CPP and CPP32.

Often, when the compiler reports an error inside a macro or an include file, you can get more information about what the error is if you can see the include files or the results of the macro expansions. In many multi-pass compilers, a separate pass performs this work, and the results of the pass can be examined. Because Borland C++ uses an integrated single-pass compiler, use CPP to get the first-pass functionality found in other compilers.

The preprocessors each have a set of command-line options which control their output. For a list of these options, type either `CPP` or `CPP32` at the command line.

For each file processed by CPP, the output is written to a file in the current directory (or the output directory named by the **-n** option) with the same name as the source name but with an extension of `.I`.

This output file is a text file containing each line of the source file and any include files. Any preprocessing directive lines have been removed, along with any conditional text lines excluded from the compile. Unless you use a command-line option to specify otherwise, text lines are prefixed with the file name and line number of the source or include file the line came from. Within a text line, any macros are replaced with their expansion text.

Note: The resulting output of CPP cannot be compiled because of the file name and line number prefix attached to each source line. However, use the **-P-** option to produce a file which doesn't have line numbers. You can then pass this file to the compiler (use the **-P** compiler option to force a C++ compile).

Using CPP

See also CPP and CPP32: The Preprocessors

Syntax

```
CPP [ options ] file[s]
```

Command-Line Options

The following options can be used with CPP:

Option	Description
-Ax	Specify language extensions
-C	Allow nested comments
-Dname	Define macro
-gnnn	Stop after N warnings
-innn	Maximum identifier length N
-Ipath	Include files directory
-jnnn	Stop after N errors
-mm	Memory Model (s is the default)
-npath	Output file directory
-ofilename	Output file name
-p	Pascal calls
-P	Include source line info (on by default)
-Uname	Undefine macro
-w	Enable all warnings
-w-xxx	Disable warning *xxx*
-wxxx	Enable warning *xxx*

CPP as a Macro Preprocessor

See also CPP and CPP32: The Preprocessors

The **-P** option tells CPP to prefix each line with the source file name and line number. With the **-P-** option, CPP can be used as a macro preprocessor; the resulting .I file can then be compiled with BCC or BCC32. (Note that you can also use the BCC option **-P** to set default file extensions.)

The following simple program illustrates how CPP preprocesses a file, first with **-P** selected, then with **-P-**.

Source file: HELLOFB.C

```
#define NAME "Frank Borland"
#define BEGIN {
#define END   }

main()
BEGIN
  printf("%s\n", NAME);
END
```

CPP command line: `CPP HELLOFB.C`

Output:

```
HELLOAJ.c 1:
HELLOAJ.c 2:
HELLOAJ.c 3:
HELLOAJ.c 4:
HELLOAJ.c 5: main()
HELLOAJ.c 6: {
HELLOAJ.c 7:    printf("%s\n","Frank Borland");
HELLOAJ.c 8: }
```

CPP command line: `CPP -P- HELLOFB.C`

Output:

```
main()
{
  printf("%s\n","Frank Borland");
}
```

Using MIDL with CPP.EXE and CPP32.EXE

See also CPP and CPP32: The Preprocessors

MIDL (Microsoft Interface Definition Language) is an RPC compiler. In order to use MIDL, with the Borland C++ preprocessors, you must use the following MIDL command:

```
MIDL -cpp_cmd {CPP|CPP32} -cpp_opt "-P- -oCON {CPP options}" {MIDL options}
  {.idl/.acf}
```

Option	Description
-*cpp_cmd* {CPP|CPP32}	Tells MIDL which preprocessor to use when processing an .IDL or .ACF file. MIDL calls the preprocessor to expand macros within source files.
-*cpp_opt* "{*options*}"	Specifies the command- line options for the preprocessor. The **-P-** removes line number and file name information from each line of the preprocessed output. The **-oCON** indicates that preprocessed output should go to standard output, instead of to file. The preprocessor banner and the current file that is being processed are not emitted. Including **-oCON** within a .CFG file processed by the preprocessor causes the banner to be emitted.
{*CPP options*}	Passes the options to CPP.
{*MIDL options*}	Any MIDL command-line options.
{*.idl/.acf file*}	The source file that MIDL processes.

CPP and UUIDs

See also CPP and CPP32: The Preprocessors

In some cases, CPP does not accept valid UUIDs. For example, a valid UUID statement is:

```
uuid(5630EAA0-CA48-1067-B320-00DD010662DB)
```

When CPP or CPP32 encounter 5630EAA0, it is classified as a floating-point number, and since it is an invalid floating point number, the preprocessor emits an error. To work around this problem, enclose the UUID within quotes and use the **-ms_ext** MIDL option. The UUID statement becomes:

```
uuid("5630EAA0-CA48-1067-B320-00DD010662DB")
```

and the MIDL command line becomes:

```
MIDL -ms_ext -cpp_cmd {CPP|CPP32} -cpp_opt "-P- -oCON
```

{CPP options}" {MIDL options} {.idl/.acf file}

GREP: A Text-Search Utility

See Also

GREP (Global Regular Expression Print) is a powerful text-search program derived from the UNIX utility of the same name. GREP searches for a text pattern in one or more files or in its standard input stream.

Here is a quick example of a situation where you might want to use GREP. Suppose you wanted to find out which text files in your current directory contained the string "Bob". You would type:

```
grep Bob *.txt
```

GREP responds with a list of the lines in each file (if any) that contained the string "Bob". Because GREP does not ignore case by default, the strings "bob" and "boB" do not match.

GREP can do a lot more than match a single, fixed string. In the following section, you will see how to make GREP search for any string that matches a particular pattern.

GREP: Command-Line Syntax

See Also

The general command-line syntax for GREP is

```
grep [-options] searchstring [file(s) ... ]
```

Option	Description
options	consist of one or more letters, preceded by a hyphen (-), that changes the behavior of GREP.
searchstring	gives the pattern to search for.
file(s)	tells GREP which files to search. (If you do not specify a file, GREP searches standard input; this lets you use pipes and redirection.)

The command `GREP ?` prints a help screen showing the options, special characters, and defaults for GREP.

Redirecting Output from GREP

If you find that the results of your GREP are longer than one screen, you can redirect the output to a file. For example, you could use this command:

```
GREP "Bob" *.txt > temp.txt
```

which searches all files with the TXT extension in the current directory, then puts the results in a file called TEMP.TXT. (You can name this file anything you like.) Use any word processor to read TEMP.TXT (the results of the search).

GREP: Command-Line Options

See Also

You can pass options to the GREP utility on the command line by specifying one or more single characters preceded by a hyphen (-). Each individual character is a switch that you can turn on or off: A plus symbol (+) after a character turns the option on; a hyphen (-) after the character turns the option off. The + sign is optional; for example, -r means the same thing as -r+. You can list multiple options individually (like this: -i -d -l), or you can combine them (like this: -ild or -il, -d, and so on).

Here are the GREP option characters and their meanings:

Option	Description
-c	Count only: Prints only a count of matching lines. For each file that contains at least one matching line, GREP prints the file name and a count of the number of matching lines. Matching lines are not printed. This option is off by default.
-d	Search subdirectories: For each file specified on the command line, GREP searches for all files that match the file specification, both in the directory specified and in all subdirectories below the specified directory. If you give a file without a path, GREP assumes the files are in the current directory. This option is off by default.
-i	Ignore case: GREP ignores upper/lowercase differences. When this option is on, GREP treats all letters a to z as identical to the corresponding letters A to Z in all situations. This option is off by default.
-l	List file names only: Prints only the name of each file containing a match. After GREP finds a match, it prints the file name and processing immediately moves on to the next file. This option is off by default.
-n	Line numbers: Each matching line that GREP prints is preceded by its line number. This option is off by default.
-o	UNIX output format: Changes the output format of matching lines to support more easily the UNIX style of command-line piping. All lines of output are preceded by the name of the file that contained the matching line. This option is off by default.
-r	Regular expression search: The text defined by searchstring is treated as a regular expression instead of as a literal string. This option is on by default. A regular expression is one or more occurrences of one or more characters optionally enclosed in quotes. The following symbols are treated specially: `^` start of line — `$` end of line `.` any character — `\` quote next character `*` match zero or more — `+` match one or more `[aeiou0-9]` match a, e, i, o, u, and 0-9 `[^aeiou0-9]` match all but a, e, i, o, u, and 0-9
-u	Update options: GREP adds any options from the command line to its default options in GREP.COM. This option lets you to customize the default option settings. To see the defaults are set in GREP.COM, type GREP ?, then each option on the help screen is followed by a + or a - depending on its default setting.
-v	Nonmatch: Prints only nonmatching lines. Only lines that do not contain the search string are considered nonmatching lines. This option is off by default.
-w	Word search: Text found that matches the regular expression is considered a match only if the character immediately preceding and following cannot be part of a word. The default word character set includes A to Z, 0 to 9, and the underscore (_). This option is off by default. An alternate form of this option lets you specify the set of legal word characters. Its form is **-w**[set], where set is any valid regular expression. If you define the set with alphabetic characters, it is automatically defined to contain both the uppercase and lowercase values for each letter in the set (regardless of how it is typed), even if the search is case-sensitive. If you use the -w option in combination with the -u option, the new set of legal characters is saved as the default set.
-z	Verbose: GREP prints the file name of every file searched. Each matching line is preceded by its line number. A count of matching lines in each file is given, even if the count is zero. This option is off by default.
?	Displays a help screen showing the options, special characters, and defaults for GREP.

GREP: The Search String

See Also

The value of searchstring defines the pattern GREP searches for. A search string can be either a regular expression or a literal string.

- In a regular expression, certain characters have special meanings: They are operators that govern the search.
- In a literal string, there are no operators: Each character is treated literally.

You can enclose the search string in quotation marks to prevent spaces and tabs from being treated as delimiters. The text matched by the search string cannot cross line boundaries; that is, all the text necessary to match the pattern must be on a single line.

A regular expression is either a single character or a set of characters enclosed in brackets. A concatenation of regular expressions is a regular expression.

When you use the **-r** option (on by default), the search string is treated as a regular expression (not a literal expression). The following characters have special meanings:

Symbol	Description
^	A circumflex at the start of the expression matches the start of a line.
$	A dollar sign at the end of the expression matches the end of a line.
.	A period matches any character.
*****	An asterisk after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, `bo*` matches `bot`, `boo`, and as well as `bo`.
+	A plus sign after a string matches any number of occurrences of that string followed by any characters, except zero characters. For example, `bo+` matches `bot` and `boo`, but not `b` or `bo`.
{ }	Characters or expressions in braces are grouped so that the evaluation of a search pattern can be controlled and so grouped text can be referred to by number.
[ ]	Characters in brackets match any one character that appears in the brackets, but no others. For example `[bot]` matches `b`, `o`, or `t`.
[^]	A circumflex at the start of the string in brackets means NOT. Hence, `[^bot]` matches any characters except `b`, `o`, or `t`.
[-]	A hyphen within the brackets signifies a range of characters. For example, `[b-o]` matches any character from `b` through `o`.
****	A backslash before a wildcard character tells the C++ IDE to treat that character literally, not as a wildcard. For example, **\^** matches ^ and does not look for the start of a line.

Four of the "special" characters ($, ., *, and +) do not have any special meaning when used within a bracketed set. In addition, the character ^ is only treated specially if it immediately follows the beginning of the set definition (immediately after the [delimiter).

GREP: File Specifications

See Also

The files option tells GREP which files to search. Files can be an explicit file name or a generic file name incorporating the DOS ? and * wildcards. In addition, you can type a path (drive and directory information). If you list files without a path, GREP searches the current directory. If you do not specify any files, input to GREP must come from redirection (<) or a vertical bar (|).

GREP: Examples

See Also

These examples show how to combine the features of GREP to do different kinds of searches. They assume default settings for GREP are unchanged.

```
grep -r [^a-z]main\ *( *.c
grep -ri [a-c]:\\data\.fil *.c *.inc
grep "search string with spaces" *.doc *.c
grep -rd "[ ,.:?'\"]"$ \*.doc
grep -w[=] = *.c
```

GREP: Example 1

See Also

Command

```
grep -r [^a-z]main\ *( *.c
```

Matches

```
main(i,j:integer)
if (main  ()) halt;
if (MAIN  ()) halt;
```

Does Not Match

```
mymain()
```

Explanation

The search string tells GREP to search for the word main with no preceding lowercase letters ([^a-z]), followed by zero or more occurrences of blank spaces (\ *), then a left parenthesis. Since spaces and tabs are normally considered command-line delimiters, you must quote them if you want to include them as part of a regular expression. In this case, the space after main is quoted with the backslash escape character. You could also accomplish this by placing the space in double quotes.

GREP: Example 2

See Also

Command

```
grep -ri [a-c]:\\data\.fil *.c *.inc
```

Matches

```
A:\data.fil
B:\DATA.FIL
c:\Data.Fil
```

Does Not Match

```
d:\data.fil
a:data.fil
```

Explanation

Because the backslash (\) and period (.) characters usually have special meaning in path and file names, you must place the backslash escape character immediately in front of them if you want to search for them. The -i option is used here, so the search is not case sensitive.

GREP: Example 3

See Also

Command

```
grep "search string with spaces" *.doc *.c
```

Matches

This is a search string with spaces in it.

Does Not Match

This search string has spaces in it.

Explanation

This is an example of how to search for a string with embedded spaces.

GREP: Example 4

See Also

Command

```
grep -rd "[ ,.:?'\"]"$ \*.doc
```

Matches

```
He said hi to me.
Where are you going?
In anticipation of a unique situation,
Examples include the following:
"Many men smoke, but fu man chu."
```

Does Not Match

```
He said "Hi" to me
Where are you going? I'm headed to the
```

Explanation

This example searches for any one of the characters " . : ? ' and , at the end of a line. The double quote within the range is preceded by an escape character so it is treated as a normal character instead of as the ending quote for the string. Also, the $ character appears outside of the quoted string. This demonstrates how regular expressions can be concatenated to form a longer expression.

GREP: Example 5

See Also

Command

```
grep -w[=] = *.c
```

Matches

```
i = 5;
j=5;
i += j;
```

Does Not Match

```
if (i == t) j++;
/* ================== */
```

Explanation

This example redefines the current set of legal characters for a word as the assignment operator (=) only, then does a word search. It matches C assignment statements, which use a single equal sign (=), but not equality tests, which use a double equal sign (==).

Using IdeToMak.EXE

IdeToMake is a utility that converts Borland C++ 5.0 project files into a C++Builder project makefile equivalent. However, because Borland C++ .IDE project files are capable of describing more complex projects than what can be used by C++Builder, the conversion utility will create a best case makefile using the following guidelines:

- C++Builder project files can contain only a single target, while Borland C++ projects can contain multiple targets. Because of this, IdeToMak will generate a separate .MAK file for each target contained in a Borland C++ project file.
- The libraries linked into the C++Builder targets will be changed to use the C++Builder multithreaded libraries. Specifically, the RTL being linked will be either cp32mt.lib or cp32mti.lib.
- Applications that use the OWL libraries will have their libraries changed to use owlwv.lib or owlwvi.lib. You must obtain these libraries from Borland C++ version 5.2. Similarly BIDS libraries will be changed to reference bidsv.lib or bidsvi.lib.
- Any references to environment variables in the include path, library path, or defines will be expanded at the time the project file is converted. This is particularly important for the environment variable BCROOT. If the variable is not in your environment, you may find that a generated makefile will have invalid entries. IDETOMAK will issue a warning if it encounters this situation.
- C++Builder does not have the ability to handle options set on individual files. Any local option overrides on nodes in the project file are ignored and a warning is presented.
- Sourcepools in project files are collapsed. There will be no notion of the sourcepool in the C++Builder makefile.
- For projects that contain multiple source modules the main source module, called the project source, will be modified to include a series of macros which tell the C++Builder make system to include those .obj files in the make. The project source will be assumed to be a .cpp file with the same name as the .ide file. If no such file exists, this file will be generated automatically so that all linked .obj files will be included. If the file does exist, it will be backed up before being modified by IDETOMAK.EXE. All additions to the file will occur at the top of the file.
- For projects that use resource files, the project source module will be similarly modified to include special macros to include the resources in C++Builder.
- The linker setting for the C++Builder makefiles generated will default to TLINK32.
- The include path added for compiler options will have `$(BCB)\include;$(BCB)\include\vcl` prepended to the path set in the original project file.
- The library path added for linker options will have `$(BCB)\lib;$(BCB)\lib\objs` prepended to the path set in the original project file.

Using IMPLIB

See also

The IMPLIB utility creates import library. IMPLIB takes as input DLLs, module definition files, or both, and produces an import library as output.

If you've created a Windows application, you've already used at least one import library, IMPORT32.LIB, the import library for the standard Windows DLLs. (IMPORT.LIB is linked automatically when you build a Windows application in the IDE and when using BCC32 to link. You have to explicitly link with IMPORT32.LIB only if you're using TLINK32 to link separately.)

An import library lists some or all of the exported functions for one or more DLLs. IMPLIB creates an import library directly from DLLs or from module definition files for DLLs (or a combination of the two).

Creating an Import Library for a DLL

Options must be lowercase and preceded by either a hyphen or a slash.

Type:

```
IMPLIB Options LibName [ DefFiles... | DLLs... ] [@ResponseFile]
```

where Options is an optional list of one or more IMPLIB options, LibName is the name for the new import library, DefFiles is a list of one or more existing module definition files for one or more DLLs, and DLLs is a list of one or more existing DLLs. You must specify at least one DLL or module definition file.

You can also use a response file to list the .DEF and .DLL files that you want to process. A response file is an ACSII text file that contains a list of files. The files must be separated by either spaces or new lines in the file. To specify a response file on the command line, precede the response filename with an "at" sign (@). For example,

```
implib  foo.lib  @respon.txt
```

Note: A DLL can also have an extension of .EXE or .DRV, not just .DLL.

Option	Description
-c	Warnings on case sensitive symbols
-f	Force imports by name
-i	Tells IMPLIB to ignore WEP, the Windows exit procedure required to end a DLL. Use this option if you are specifying more than one DLL on the IMPLIB command line.
-o	Remove module extensions. (16-bit Windows only)
-w	No warnings.

Using the module definition file manager (IMPDEF)

See also

Import libraries provide access to the functions in a Windows DLL.

IMPDEF takes as input a DLL name, and produces as output a module definition file with an export section containing the names of functions exported by the DLL.

The syntax is:

```
IMPDEF DestName.DEF SourceName.DLL
```

This creates a module definition file named DestName.DEF from the file SourceName.DLL. The resulting module definition file would look something like this:

```
LIBRARY     FileName
DESCRIPTION 'Description'
EXPORTS
            ExportFuncName              @Ordinal
         .
         .
         .
            ExportFuncName              @Ordinal
```

where:

- FileName is the DLL's root file name
- Description is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement
- ExportFuncName names an exported function
- Ordinal is that function's ordinal value (an integer).

Classes in a DLL

See also

IMPDEF is useful for a DLL that uses C++ classes. If you use the _export keyword when defining a class, all of the non-inline member functions and static data members for that class are exported. It's easier to let IMPDEF make a module definition file for you because it lists all the exported functions, and automatically includes the member functions and static data members.

Since the names of these functions are mangled, it would be tedious to list them all in the EXPORTS section of a module definition file simply to create an import library from the module definition file. If you use IMPDEF to create the module definition file, it will include the ordinal value for each exported function. If the exported name is mangled, IMPDEF will also include that function's unmangled, original name as a comment following the function entry. So, for instance, the module definition file created by IMPDEF for a DLL that used C++ classes would look something like this:

```
LIBRARY     FileName
DESCRIPTION 'Description'
EXPORTS
            MangledExportFuncName  @Ordinal ; ExportFuncName
        .
        .
        .
            MangledExportFuncName  @Ordinal ; ExportFuncName
```

where

- FileName is the DLL's root file name
- Description is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement
- MangledExportFuncName provides the mangled name
- Ordinal is that function's ordinal value (an integer)
- ExportFuncName gives the function's original name.

Functions in a DLL

See also

IMPDEF creates an editable source file that lists all the exported functions in the DLL. You can edit this .DEF file to contain only those functions that you want to make available to a particular application, then run IMPLIB on the edited .DEF file. This results in an import library that contains import information for a specific subset of a DLL's export functions.

Suppose you're distributing a DLL that provides functions to be used by several applications. Every export function in the DLL is defined with _export. Now, if all the applications used all the DLL's exports, then you could use IMPLIB to make one import library for the DLL. You could deliver that import library with the DLL, and it would provide import information for all of the DLL's exports. The import library could be linked to any application, thus eliminating the need for the particular application to list every DLL function it uses in the IMPORTS section of its module definition file.

But let's say you want to give only a few of the DLL's exports to a particular application. Ideally, you want a customized import library to be linked to that application--an import library that provides import information only for the subset of functions that the application will use. All of the other export functions in the DLL will be hidden to that client application.

To create an import library that satisfies these conditions, run IMPDEF on the compiled and linked DLL. IMPDEF produces a module definition file that contains an EXPORT section listing all of the DLL's export functions. You can edit that module definition file, remove the EXPORTS section entries for those functions you don't want in the customized import library, and then run IMPLIB on the module definition file. The result will be an import library that contains import information for only those export functions listed in the EXPORTS section of the module definition file.

Using TLIB

See also

When it modifies an existing library, TLIB always creates a copy of the original library with a .BAK extension.

Build or Modify Libraries

The libraries included with Borland C++ were built with TLIB. You can use TLIB to build your own libraries, or to modify the Borland C++ libraries, your libraries, libraries furnished by other programmers, or commercial libraries you've purchased.

You can use TLIB to:

- Create a new library from a group of object modules.
- Add object modules or other libraries to an existing library.
- Remove object modules from an existing library.
- Replace object modules from an existing library.
- Extract object modules from an existing library.
- List the contents of a new or existing library.

TLIB can also create (and include in the library file) an extended dictionary, which can be used to speed up linking.

Although TLIB is not essential for creating executable programs with Borland C++, it is a useful programming productivity tool that can be indispensable for large development projects.

TLIB Command-line Options

See also

The TLIB command line takes the following general form, where items listed in square brackets are optional:

```
tlib [@respfile] [option] libname [operations] [, listfile]
```

TLIB options

For an online summary of TLIB options, type TLIB and press Enter.

Option	Description
@respfile	The path and name of the response file you want to include. You can specify more than one response file.
libname	The DOS path name of the library you want to create or manage. Every TLIB command must be given a libname. Wildcards are not allowed. TLIB assumes an extension of .LIB if none is given. Use only the .LIB extension because both BCC and the IDE require the .LIB extension in order to recognize library files. Note: If the named library does not exist and there are add operations, TLIB creates the library.
/C	The case-sensitive flag. This option is not normally used.
/E	Creates Extended Dictionary
/Psize	Sets the library page size to size.
/0	Purges comment records.
operations	The list of operations TLIB performs. Operations can appear in any order. If you only want to examine the contents of the library, don't give any operations.
listfile	The name of the file that lists library contents. The listfile name (if given) must be preceded by a comma. No listing is produced if you don't give a file name. The listing is an alphabetical list of each module. The entry for each module contains an alphabetical list of each public symbol defined in that module. The default extension for the listfile is .LST. You can direct the listing to the screen by using the listfile name CON, or to the printer by using the name PRN.

Using TLIB Response Files

See also

When you use a large number of operations, or if you find yourself repeating certain sets of operations over and over, you will probably want to use response files. A response file is an ASCII text file (which can be created with the Borland C++ editor) that contains all or part of a TLIB command. Using TLIB response files, you can build TLIB commands larger than would fit on one command line. Response files can

- Contain more than one line of text; use the ampersand character (&) at the end of a line to indicate that another line follows.
- Include a partial list of commands. You can combine options from the the command line with options in a response file.
- be used with other response files in a single TLIB command line.

TLIB /C Option

See also

Using Case-Sensitive Symbols in a Library

When you add a module to a library, TLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add a module to the library that duplicates a symbol, TLIB displays an error message and doesn't add the module.

Normally, when TLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not treated differently (for example, the symbols lookup and LOOKUP are treated as duplicates). You can use the /C option to add a module to a library that includes symbols differing only in case.

Don't use /C if you plan to use the library with other linkers or let other people use the library.

TLIB normally rejects symbols that differ only in case because some linkers aren't case-sensitive. TLINK has no problem distinguishing uppercase and lowercase symbols. As long as you use your library only with TLINK, you can use the TLIB /C option without any problems.

TLIB /E Option

See also

Creating an Extended Dictionary

To increase the capacity of TLINK for large links, you can use TLIB to create an extended dictionary and append it to the library file. This dictionary contains, in a compact form, information that is not included in the standard library dictionary and that lets TLINK process library files so that those modules not needed in the link are not processed.

To create an extended dictionary for a library that is being modified, use the /E option when you start TLIB to add, remove, or replace modules in the library. To create an extended dictionary for an existing library that you don't want to modify, use the /E option. For example, if you type the following text, TLINK appends an extended dictionary to the specific library:

```
tlib /E mylib
```

If you get the message "Table limit exceeded", use /E to see if it helps. If you use /E to add a library module containing a C++ class with a virtual function, you'll get the error message, Library contains COMDEF records--extended dictionary not created.

TLIB /P Option

See also

Setting the Page Size to Create a Large Library

Every DOS library file contains a dictionary that appears at the end of the .LIB file, following all of the object modules. For each module in the library, the dictionary contains a 16-bit address of that particular module within the .LIB file; this address is given in terms of the library page size (it defaults to 16 bytes).

The library page size determines the maximum combined size of all object modules in the library, which cannot exceed 65,536 pages. The default (and minimum) page size of 16 bytes allows a library of about 1 MB in size. To create a larger library, use the /P option to increase the page size. The page size must be a power of 2, and it cannot be smaller than 16 or larger than 32,768.

All modules in the library must start on a page boundary. For example, in a library with a page size of 32 (the lowest possible page size higher than the default 16), an average of 16 bytes will be lost per object module in padding. If you attempt to create a library that is too large for the given page size, TLIB will issue an error message and suggest that you use /P with the next available higher page size.

Operation List

See also

The operation list describes what actions you want TLIB to do and consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character action symbol followed by a file or module name. You can put whitespace around either the action symbol or the file or module name, but not in the middle of a two-character action or in a name.

You can put as many operations as you like on the command line, up to DOS's COMMAND.COM-imposed line-length limit of 127 characters. The order of the operations is not important. TLIB always applies the operations in a specific order:

To replace a module, remove it, then add the replacement module.

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.

TLIB finds the name of a module by stripping any drive, path, and extension information from the given file name.

Note that TLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you need to supply only the module name, not the path and .OBJ extension.

Wildcards are never allowed in file or module names.

TLIB recognizes three action symbols (*, +, *), which you can use singly or combined in pairs for a total of five distinct operations. The order of the characters is not important for operations that use a pair of characters. The action symbols and what they do are listed here:

To create a library, add modules to a library that does not yet exist.

TLIB action symbols

Symbol	Name	Description
+	Add	TLIB adds the named file to the library. If the file has no extension, TLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library. If a module being added already exists, TLIB displays a message and does not add the new module.
-	Remove	TLIB removes the named module from the library. If the module does not exist in the library, TLIB displays a message. A remove operation needs only a module name. TLIB lets you enter a full path name with drive and extension included, but ignores everything except the module name.
*****	Extract	TLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten. You can't directly rename modules in a library. To rename a module, extract and remove it, rename the file just created, then add it back into the library.
-*	Extract &	TLIB copies the named module to the corresponding
***-**	Remove	file name and then removes it from the library.
-+	Replace	TLIB replaces the named module with the corresponding file.

TLIB Examples

These simple examples demonstrate some of the different things you can do with TLIB:

Example 1

To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type:

```
tlib mylib +x +y +z.
```

Example 2

To create a library named MYLIB.LIB and get a listing in MYLIB.LST too, type:

```
tlib mylib +x +y +z, mylib.lst.
```

Example 3

To get a listing in CS.LST of an existing library CS.LIB, type:

```
tlib cs, cs.lst.
```

Example 4

To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type:

```
tlib mylib -+x +a -z.
```

Example 5

To extract module Y.OBJ from MYLIB.LIB and get a listing in MYLIB.LST, type:

```
tlib mylib *y, mylib.lst.
```

Example 6

To create a new library named ALPHA, with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:

1. First create a text file, ALPHA.RSP, with

```
+a.obj +b.obj +c.obj &
      +d.obj +e.obj +f.obj &
      +g.obj
```

2. Then use the TLIB command, which produces a listing file named ALPHA.LST:

```
tlib alpha @alpha.rsp, alpha.lst
```

IMPLIB

IMPLIB creates import libraries, and IMPDEF creates module definition files (.DEF files). Import libraries and module definition files provide information to the linker about functions imported from dynamic-link libraries (DLLs).

TLIB

TLIB is a utility that manages libraries of individual .OBJ (object module) files. A library is a convenient tool for dealing with a collection of object modules as a single unit.

Import Libraries

Import libraries contain records. Each record contains the name of a DLL, and specifies where in the DLL the imported functions reside. These records are bound to the application by TLINK or the IDE linker, and provide Windows with the information necessary to resolve DLL function calls. An import library can be substituted for part or all of the IMPORTS section of a module definition file.

IMPDEF

IMPDEF takes as input a DLL name, and produces as output a module definition file with an export section containing the names of functions exported by the DLL.

About MAKE

See also

MAKE.EXE is a command-line utility that helps you manage project compilation and link cycles. MAKE is not inherently tied to compiling and linking, but is a more generic tool for executing commands based on file dependencies. MAKE helps you quickly build projects by compiling only the files you have modified since the last compilation. In addition, you can set up rules that specify how MAKE should deal with the special circumstances in your builds.

MAKE Basics

MAKE uses rules you write along with its default settings to determine how it should compile the files in your project. For example, you can specify when to build your projects with debug information and to compile your .OBJ files only if the date/time stamps of a source file is more recent than the .OBJ itself. If you need to force the compilation of a module, use TOUCH.EXE to modify the time stamp of one of the module's dependents.

In an ASCII *makefile*, you write explicit and implicit rules to tell MAKE how to treat the files in your project; MAKE determines if it should execute a command on a file or set of files using the rules you set up. Although your commands usually tell MAKE to compile or link a set of files, you can specify nearly any operating system command with MAKE.

The general syntax for MAKE is

```
MAKE [options...] [target[target]]
```

options
are MAKE options that control how MAKE works

target
is the name of the target listed in the makefile that you want to build

You must separate the MAKE command and the *options* and *target* arguments with spaces. When specifying targets, you can use wildcard characters (such as * and ?) to indicate multiple files. To get command-line help for MAKE, type `MAKE -?`.

Note: If you need to compile in real mode, use the program MAKER.EXE.

Default MAKE actions

When you issue a MAKE command, MAKE looks for the file BUILTINS.MAK, which contains the default rules for MAKE (use the **-r** option to ignore the default rules). MAKE looks for this file first in the current directory, then in the directory where MAKE.EXE is stored. After loading BUILTINS.MAK, MAKE looks in the current directory for a file called MAKEFILE or MAKEFILE.MAK (use the **-f** option to specify a file other than MAKEFILE). If MAKE can't find either of these files, it generates an error message.

After loading the makefile, MAKE tries to build only the first explicit target listed in the makefile by checking the time and date of the dependent files of the first target. If the dependent files are more recent than the target file, MAKE executes the commands to update the target.

If one of the first target's dependent files is used as a target elsewhere in the makefile, MAKE checks that target's dependencies and builds it before building the first target. This chain reaction is called a *linked dependency*.

If something during the build process fails, MAKE deletes the target file it was building. Use the .precious directive if you want MAKE to keep the target when a build fails.

You can stop MAKE after issuing the MAKE command by pressing Ctrl+Break or Ctrl+C.

BUILTINS.MAK

The file BUILTINS.MAK contains standard rules and macros that MAKE uses when it builds the targets in a makefile. To ignore this file, use the **-r** MAKE option.

Here is the default text of BUILTINS.MAK:

```
#
# <Default ¶ Font>Borland C++ - (C) Copyright 1993 by Borland International
#

# default is to target 16BIT
# pass -DWIN32 to make to target 32BIT

!if !$d(WIN32)
CC        = bcc
RC        = brcc
AS        = tasm
!else
CC        = bcc32
RC        = brcc32
AS        = tasm32
!endif
.asm.obj:
    $(AS) $(AFLAGS) $&.asm
.c.exe:
    $(CC) $(CFLAGS) $&.c
.c.obj:
    $(CC) $(CFLAGS) /c $&.c
.cpp.exe:
    $(CC) $(CFLAGS) $&.cpp
.cpp.obj:
    $(CC) $(CPPFLAGS) /c $&.cpp
.rc.res:
    $(RC) $(RFLAGS) /r $&

.SUFFIXES: .exe .obj .asm .c .res .rc

!if !$d(BCEXAMPLEDIR)
BCEXAMPLEDIR = $(MAKEDIR)\..\EXAMPLES
!endif
```

About makefiles

A *makefile* is an ASCII file that contains the set of instructions that MAKE uses to build a certain project. Although MAKE assumes your makefile is called MAKEFILE or MAKEFILE.MAK, you can specify a different makefile name with the **-f** option.

MAKE either builds the target(s) you specify with the make command or it builds the first target it finds in the makefile To build more than a single target, use a symbolic target in your makefile.

Makefiles can contain

- Comments (precede with a number sign [#])
- Explicit and implicit rules
- Macros
- Directives

Symbolic targets

A *symbolic target* forces MAKE to build multiple targets in a makefile. When you specify a symbolic target, the dependency line lists all the targets you want to build (a symbolic target basically uses linked dependencies to build more than one target).

For example, the following makefile uses the symbolic target AllFiles to build both FILE1.EXE and FILE2.EXE:

```
AllFiles: file1.exe file2.exe  #Note that AllFiles has no commands
file1.exe: file1.obj
    bcc file1.obj
file2.exe: file2.obj
    bcc file2.obj
```

Rules for symbolic targets

Observe the following rules when you use symbolic targets:

- Do not type a line of commands after the symbolic target line.
- A symbolic target must have a unique name; it cannot be the name of a file in your current directory.
- Symbolic target names must follow the operating system rules for naming files.

MAKE options

You can use command-line options to control the behavior of MAKE. MAKE options are case-sensitive and must be preceded with either a hyphen (-) or slash (/).

The general syntax for MAKE is

```
MAKE [options...] [target[target]]
```

options
are MAKE options that control how MAKE works

target
is the name of the target listed in the makefile that you want to build

You must separate the MAKE command and the *options* and *target* arguments with spaces. When specifying targets, you can use wildcard characters (such as * and ?) to indicate multiple files. To get command-line help for MAKE, type `MAKE -?`.

For example, to use a file called PROJECTA.MAK as the makefile, type `MAKE -fPROJECTA.MAK`. Many of the command-line options have equivalent directives that you can use within the makefile.

- Use the **-W** option to set default MAKE options.

Option	Description
-a	Checks dependencies of include files and nested include files associated with .OBJ files and updates the .OBJ if the .h file changed. See also **-c**.
-B	Builds all targets regardless of file dates.
-c	Caches autodependency information, which can improve MAKE speed. Use with **-a**. Do not use this option if MAKE modifies include files (which can happen if you use TOUCH in the makefile or if you create header or include files during the MAKE process).
-D*macro*	Defines *macro* as a single character, causing an expression **!ifdef** macro written in the makefile to return **true**.
[**-D**]*macro*=[*string*]	Defines *macro* as *string*. If *string* contains any spaces or tabs, enclose *string* in quotation marks. The **-D** is optional.
-d*directory*	Specifies the drive and directory that MAKER (the real mode version of MAKE) uses when it swaps out of memory. This option must be used with **-S**. MAKE ignores this option.
-e	Ignores a macro if its name is the same as an environment variable (MAKE uses the environment variable instead of the macro).
-f*filename*	Uses *filename* or *filename.MAK* instead of MAKEFILE (a space after **-f** is optional).
-h or **-?**	Displays MAKE options. Default settings are shown with a trailing plus sign.
-I*directory*	Searches for include files in the current directory first, then in *directory* you specify with this option.
-i	Ignores the exit status of all programs run from the makefile and continues the build process.
-K	Keeps temporary files that MAKE creates (MAKE usually deletes them).
-m	Displays the date and time stamp of each file as MAKE processes it.
-N	Causes MAKE to mimic Microsoft's NMAKE.
-n	Prints the MAKE commands but does not perform them, this is helpful for debugging makefiles.
-p	Displays all macro definitions and implicit rules before executing the makefile.
-q	Returns 0 if the target is up-to-date and nonzero if it is not (for use with batch

	files).
-r	Ignores any rules defined in BUILTINS.MAK.
-S	Swaps MAKER out of memory while commands are executed, reducing memory overhead and allowing compilation of large modules. MAKE ignores this option.
-s	Suppresses onscreen command display.
-U*macro*	Undefines the previous macro definition of *macro*.
-W	Writes the specified non-string options to MAKE.EXE, making them defaults.

Setting default MAKE options

The **-W** option lets you set the default options for MAKE. Use the following syntax to set the default options:

```
make [-option[-] ...] -W
```

For example, type

```
MAKE -m -W
```

to turn the **-m** option on by default (which causes MAKE to always display file dates and times). When you use the **-W** option, MAKE asks you to write changes to MAKE.EXE. Type Y to accept the new defaults. To turn off an option that's on by default, follow the option with a hyphen. For example, to undo the **-m** option change, type

```
MAKE -m- -W
```

- The **-W** option doesn't work with the following MAKE options:

-D*macro*	-D*macro=string*
-d*directory*	-U*symbol*
-f*filename*	-? or -h
-I*directory*	

Note: If you attempt to use the **-W** option when the DOS SHARE program is loaded, MAKE displays the message `Fatal: unable to open file MAKE.EXE`.

Using temporary response files

MAKE can create temporary response files when your command lines become too long to place on a single line.

To begin writing to a response file, place the MAKE operator && followed by a delimiter of your choice (| makes a good delimiter) in the makefile. To finish writing to the file, repeat your delimiter.

The following example shows `&&|` instructing MAKE to create a file for the input to TLINK32.

```
prog.exe: A.obj B.obj
    TLINK32 /c @&&|     # &&| opens temp file, @ for TLINK32
    c0s.obj $**
    prog.exe
    prog.map
    maths.lib cs.lib
|                       # | closes temp file, must be on first column
```

The response file created by `&&|` contains these instructions:

```
c0s.obj a.obj b.obj
prog.exe
prog.map
maths.lib cs.lib
```

MAKE names temporary file starting at MAKE0000.@@@, where the 0000 increments by one with each temporary file you create. MAKE later deletes the temporary file when it terminates.

Using TOUCH

TOUCH.EXE updates a file's date stamp so that it reflects your system's current time and date.

Sometimes you might need to force a target to be recompiled or rebuilt even though you haven't changed its source files. One way to do this is to use the TOUCH utility to update the time stamp of one or more of the target's dependency files. To touch a file (or files), type the following at the command prompt:

```
touch [options] filename [filename...]
```

Because TOUCH is a 32-bit executable, it accepts long file names. In addition, you can use file names that contain the wildcard characters * and ? to "touch" more than a single file at a time.

Note: Before you use TOUCH, make sure your system's internal clock is set correctly.

TOUCH options

TOUCH.EXE supports several command-line options:

Option	Description
-d*mm-dd-yy*	Sets the date of the file to the specified date.
-f*filename*	Sets the time and date of files to match those of *filename*.
-h	Displays help information (same as typing TOUCH without options or file names).
-t*hh:mm:ss*	Sets the time of the file to the specified time.
-v	Verbose mode, shows each file TOUCHed.

Compatibility with Microsoft's NMAKE

Use the **-N** option if you want to use a makefile that was originally created for Microsoft's NMAKE. The following changes occur when you use **-N**:

- The **$d** macro is treated differently-use **!ifdef** or **!ifndef** instead.
- Macros that return paths won't return the last \. For example, if **$(<D)** normally returns C:\CPP\, the **-N** option causes MAKE to return C:\CPP.
- Unless there's a matching **.suffixes** directive, MAKE begins searching for rules from the bottom of the makefile and works its way to the top.
- In implicit rules, MAKE expands **$*** macros to the target name instead of to the dependent name.
- MAKE interprets the << operator to generate temporary files, much as it would for the && operator. MAKE uses temporary files as response files. These files are then deleted. To keep a file, either use the **-K** MAKE command-line option or use **KEEP** in the makefile.

```
<<FileName.Ext
    text
    ...
    <<KEEP
```

If you don't want to keep a temporary file, type NOKEEP or type only the temporary (optional) file name. If you don't type a file name, MAKE creates a name for you. If you use **NOKEEP**, it will override the **-K** command-line option.

Explicit and implicit rules

You write *explicit and implicit rules* to instruct MAKE how to build the targets in your makefile. In general, these rules are defined as follows:

Explicit rules are instructions for specific files.

Implicit rules are general instructions for files without explicit rules.

All the rules you write follow this general format:

```
Dependency line
    Command line
```

While the dependency line has a different syntax for explicit and implicit rules, the command line syntax stays the same for both rule types.

MAKE supports multiple dependency lines for a single target, and a single target can have multiple command lines. However, only one dependency line should contain a related command line. For example:

```
Target1: dependent1 dep2 dep3 dep4 dep5
Target1: dep6 dep7 dep8
    bcc -c $**
```

Explicit rule syntax

Explicit rules specify the instructions that MAKE must follow when it builds specific targets. Explicit rules name one or more targets followed by one or two colons. One colon means one rule is written for the target(s); two colons mean that two or more rules are written for the target(s).

Explicit rules follow this syntax:

```
target [target...]:[:][{path}] [dependent[s]...]
    [commands]
```

target
specifies the name and extension of the file to be built (a target must begin a line in the makefile-you cannot precede the target name with spaces or tabs). To specify more than one target, separate the target names with spaces or tabs. Also, you cannot use a target name more than once in the target position of an explicit rule.

path
is a list of directories that tells MAKE where to find the dependent files. Separate multiple directories with semicolons and enclosed the entire path specification in braces.

dependent
is the file (or files) whose date and time MAKE checks to see if it is newer than `target`. Each dependent file must be preceded by a space. If a dependent appears elsewhere in the makefile as a target, MAKE updates or creates that target before using the dependent in the original target (this in known as a *linked dependency*).

commands
are any operating system command or commands. You must indent the command line by at least one space or tab, otherwise they are interpreted as a target. Separate multiple commands with spaces.

If a dependency or command line continues on the following line, use a backslash (\) at the end of the first line to indicate that the line continues. For example,

```
MYSOURCE.EXE: FILE1.OBJ\          #Dependency line
    FILE3.OBJ                     #Dependency line continued
    bcc file1.obj file3.obj       #Command line
```

Single targets with multiple rules

A single target can have more than one explicit rule. To specify more than a single explicit rule, use a double colon (::) after the target name. The following example shows targets with multiple rules and commands.

```
.cpp.obj:
    bcc -c -ncobj $<

.asm.obj:
    tasm  /mx $<, asmobj\

mylib.lib :: f1.obj f2.obj      #double colon specifies multipe rules
    echo Adding C files
    tlib mylib -+cobjf1 -+cobjf2

mylib.lib :: f3.obj f4.obj
    echo Adding ASM files
    tlib mylib -+asmobjf3 -+asmobjf4
```

Implicit rule syntax

An *implicit rule* specifies a general rule for how MAKE should build files that end with specific file extensions. Implicit rules start with either a path or a period. Their main components are file extensions separated by periods. The first extension belongs to the dependent, the second to the target.

If implicit dependents are out-of-date with respect to the target, or if the dependents don't exist, MAKE executes the commands associated with the rule. MAKE updates explicit dependents before it updates implicit dependents.

Implicit rules follow this basic syntax:

```
[{source_dir}].source_ext[{target_dir}].target_ext:
    [commands]
```

source_dir
specifies the directory (or directories) containing the dependent files. Separate multiple directories with a semicolon.

.source_ext
specifies the dependent filename extension.

target_dir
specifies the directory where MAKE places the target files. The implicit rule will only be used for targets in this directory. Without specifying a target directory, targets from any directory will match the implicit rule.

.target_ext
specifies the target filename extension. Macros are allowed here.

: (colon)
marks the end of the dependency line.

commands
are any operating system command or commands. You must indent the command line by at least one space or tab, otherwise they are interpreted as a target.

If two implicit rules match a target extension but no dependent exists, MAKE uses the implicit rule whose dependent's extension appears first in the .SUFFIXES list.

Explicit rules with implicit commands

A target in an explicit rule can get its command line from an implicit rule. The following example shows an implicit rule followed by an explicit rule without a command line.

```
.c.obj:
    bcc -c $<   #This command uses a macro $< described later

myprog.obj:     #This explicit rule uses the command: bcc -c myprog.c
```

The implicit rule command tells MAKE to compile MYPROG.C (the macro **$<** replaces the name myprog.obj with myprog.c).

Command syntax

Commands immediately follow an explicit or implicit rule and must begin on a new line with a space or tab.

Commands can be any operating system command, but they can also include MAKE macros, directives, and special operators that your operating system won't recognize (however, note that | can't be used in commands). Here are some sample commands:

```
cd..

bcc -c mysource.c

COPY *.OBJ C:\PROJECTA

bcc -c $(SOURCE)     #Macros are explained later in the chapter.
```

Commands follow this general syntax:

```
[prefix...] commands
```

Command prefixes

Commands in both implicit and explicit rules can have prefixes that modify how MAKE treats the commands. The following table lists the prefixes you can use in makefiles:

Prefix	Description
@	Don't display the command while it's being executed.
-*num*	Stop processing commands in the makefile when the exit code returned from command exceeds the integer *num*. Normally, MAKE aborts if the exit code is nonzero. No space is allowed between - and *num*.
-	Continue processing commands in the makefile, regardless of the exit codes they return.
&	Expand either the macro **$****, which represents all dependent files, or the macro **$?**, which represents all dependent files stamped later than the target. Execute the command once for each dependent file in the expanded macro.
!	Will behave like the & prefix.

Using @

The following command uses the **@** prefix, which prevents MAKE from displaying the command onscreen.

```
diff.exe : diff.obj
    @bcc diff.obj
```

Using -num and -

The **-num** and **-** prefixes control the makefile processing when errors occur. You can choose to continue with the MAKE process if an error occurs or you can specify a number of errors to tolerate.

In the following example, MAKE continues processing if BCC returns errors:

```
target.exe : target.obj
target.obj : target.cpp
    -bcc -c target.cpp
```

Using &

The **&** prefix issues a command once for each dependent file. It is especially useful for commands that don't take a list of files as parameters. For example,

```
copyall : file1.cpp file2.cpp
    &copy $** c:\temp
```

invokes COPY twice as follows:

```
copy file1.cpp c:\temp
copy file2.cpp c:\temp
```

Without the **&** modifier, MAKE would call COPY only once. Note: the **&** prefix only works with $** and $! macros.

MAKE command operators

While you can use any operating system command in a MAKE command section, you can also use the following special operators:

Operator	Description
<	Use input from a specified file rather than from standard input
>	Send the output from command to file
>>	Append the output from command to file
<<	Create a temporary inline file and use its contents as standard input to command. Also, create temporary response file when **-N** is used. Note: this is only for use with NMAKE.
&&	Create a temporary response file and insert its name in the makefile
delimiter	Use delimiters with temporary response files. You can use any character other than # as a delimiter. Use << and && as a starting and ending delimiter for a temporary file. Any characters on the same line and immediately following the starting delimiter are ignored. The closing delimiter must be written on a line by itself.

Using MAKE macros

A macro is a variable that MAKE expands into a string whenever MAKE encounters the macro in a makefile. For example, you can define a macro called LIBNAME that represents the string "mylib.lib." To do this, type the line `LIBNAME = mylib.lib` at the beginning of your makefile. Then, when MAKE encounters the macro $(LIBNAME), it substitutes the string mylib.lib. Macros let you create template makefiles that you can change to suit different projects.

To use a macro in a makefile, type $(MacroName) where MacroName is a defined macro. You can use braces or parentheses to enclose *MacroName*.

MAKE expands macros at various times depending on where they appear in the makefile:

- Nested macros are expanded when the outer macro is invoked.
- Macros in rules and directives are expanded when MAKE first looks at the makefile.
- Macros in commands are expanded when the command is executed.

If MAKE finds an undefined macro in a makefile, it looks for an operating system environment variable of that name (usually defined with **SET**) and uses its definition as the expansion text. For example, if you wrote $(PATH) in a makefile and never defined PATH, MAKE would use the text you defined for PATH in your AUTOEXEC.BAT. See your operating system manuals for information on defining environment variables.

Defining MAKE macros

The general syntax for defining a macro in a makefile is:

```
MacroName = expansion_text.
```

MacroName
is case-sensitive (MACRO1 is different from Macro1).

MacroName
is limited to 512 characters.

expansion_text
is limited to 4096 characters. Expansion characters may be alphanumeric, punctuation, or spaces.

You must define each macro on a separate line in your makefile and each macro definition must start on the first character of the line. For readability, macro definitions are usually put at the top of the makefile. If MAKE finds more than one definition of MacroName, the new definition overwrites the old one.

You can also define macros using the **-D** command-line option. No spaces are allowed before or after the equal sign (=); however, you can define more than one macro by separating the definitions with spaces. The following examples show macros defined at the command line:

```
make -Dsourcedir=c:\projecta
make -Dcommand="bcc -c"
make -Dcommand=bcc option=-c
```

- Macros defined in makefiles overwrite those defined on the command line.

String substitutions in MAKE macros

MAKE lets you temporarily substitute characters in a previously defined macro. For example, if you defined the macro

```
SOURCE = f1.cpp f2.cpp f3.cpp
```

you could substitute the characters .obj for the characters .cpp by using the MAKE command `$(SOURCE:.cpp=.obj)`. This substitution does not redefine the macro.

Rules for macro substitution:

- Syntax: `$(MacroName:original_text=new_text)`
- No space before or after the colon
- Characters in original_text must exactly match the characters in the macro definition (text is case-sensitive)

MAKE also lets you use macros within substitution macros. For example,

```
MYEXT=.C
SOURCE=f1.cpp f2.cpp f3.cpp
$(SOURCE:.cpp=$(MYEXT))          #Changes f1.cpp to f1.C, etc.
```

Default MAKE macros

MAKE contains several default macros you can use in your makefiles. The following table lists the macro definition and what it expands to in explicit and implicit rules.

Macro	Expands in implicit	Expands in explicit
$*	path\dependent file	path\target file
$<	path\dependent file+ext	path\target file+ext
$:	path for dependents	path for target
$.	dependent file+ext	target file + ext
$&	dependent file	target file
$@	path\target file+ext	path\target file+ext
$**	path\dependent file+ext	all dependents file+ext
$?	path\dependent file+ext	old dependents

Macro	Expands to	Comment
_ _MSDOS_ _	1	If running under DOS
_ _MAKE_ _	0x0370	MAKE's hex version number
MAKE	make	MAKE's executable file name
MAKEFLAGS	options	The options typed on the command line
MAKEDIR	directory	Directory where MAKE.EXE is located

Modifying default MAKE macros

If the default macros don't give you the exact string you want, macro modifiers let you extract parts of the string to suit your purpose. Macro modifiers are usually used with $< or $@.

To modify a default macro, use this syntax:

```
$(MacroName [modifier])
```

The following table lists macro modifiers and provides examples of their use.

Modifier	Part of file name expanded	Example	Result
D	Drive and directory	$(<D)	C:\PROJECTA\
F	Base and extension	$(<F)	MYSOURCE.C
B	Base only	$(<B)	MYSOURCE
R	Drive, directory, and base	$(<R)	C:\PROJA\SOURCE

Using MAKE directives

MAKE directives resemble directives in languages such as C and Pascal. In MAKE, they perform various control functions, such as displaying commands onscreen before executing them. MAKE directives begin either with an exclamation point or a period, and they override any options given on the command line.

The following table lists the MAKE directives and their corresponding command-line options:

Directive	Option	Description
.autodepend	**-a**	Turns on autodependency checking
.cacheautodepend	**-c**	Turns on autodependency caching
!cmdswitches		Uses + or - followed by non-string option letters to turn each option on or off. Spaces or tabs must appear before the + or - operator, none can appear between the operator and the option letters.
!elif		Acts like a C **else if**
!else		Acts like a C **else**
!endif		Ends an **!if**, **!ifdef**, or **!ifndef** statement
!error		Stops MAKE and prints an error message
!if		Begins a conditional statement
!ifdef		Acts like a C **#ifdef**, testing whether a given macro has been defined
!ifndef		Acts like a C **#ifndef**, testing whether a given macro is undefined
.ignore	**-i**	MAKE ignores the return value of a command
!include		Acts like a C #include, specifying a file to include in the makefile
.keep	**-K**	Keeps temporary files that MAKE creates (MAKE usually deletes them)
!message		Prints a message to **stdout** while MAKE runs the makefile
.noautodepend	**-a-**	Turns off autodependency checking
.nocacheautodepend	**-c-**	Turns off autodependency caching
.noIgnore	**-i-**	Turns off **.Ignore**
.nokeep	**-K-**	Does not keep temporary files that MAKE creates
.nosilent	**-s-**	Displays commands before MAKE executes them
.noswap	**-S-**	Tells MAKE not to swap itself out of memory before executing a command
.path.ext		Tells MAKE to search for files with the extension .ext in path directories
.precious		Saves the target or targets even if the build fails
.silent	**-s**	MAKE executes commands without printing them first
.suffixes		Determines the implicit rule for ambiguous dependencies
.swap	**-S**	Tells MAKE to swap itself out of memory before executing a command
!undef		Clears the definition of a macro. After this, the macro is undefined

.autodepend

Autodependencies are the files that are automatically included in the targets you build, such as the header files included in your C++ source code. With .autodepend on, MAKE compares the dates and times of all the files used to build the .OBJ, including the autodependency files. If the dates or times of the files used to build the .OBJ are newer than the date/time stamp of the .OBJ file, the .OBJ file is recompiled. You can use .autodepend (or **-a**) in place of forming linked dependencies.

!error

The syntax of the **!error** directive is:

```
!error message
```

MAKE stops processing and prints the following string when it encounters this directive:

```
Fatal makefile exit code: Error directive: message
```

Embed **!error** in conditional statements to abort processing and print an error message, as shown in the following example:

```
!if !$d(MYMACRO)
#if MYMACRO isn't defined
!error MYMACRO isn't defined
!endif
```

If MYMACRO isn't defined, MAKE terminates and prints:

```
Fatal makefile 4: Error directive: MYMACRO isn't defined
```

Error-checking controls

MAKE offers four different controls to control error checking:

- The **.ignore** directive turns off error checking for a selected portion of the makefile.
- The **-i** command-line option turns off error checking for the entire makefile.
- The **-num** prefix, which is entered as part of a rule, turns off error checking for the related command if the exit code exceeds the specified number.
- The **-** prefix turns off error checking for the related command regardless of the exit code.

!if and other conditional directives

The **!if** directive works like C if statements. As shown here, the syntax of !if and the other conditional directives resembles compiler conditionals:

!if condition	!if condition	!if condition	!ifdef macro
!endif	!else	!elif condition	!endif
	!endif	!endif	

The following expressions are equivalent:

```
!ifdef macro and !if $d(macro)
ifndef macro and !if !$d(macro)
```

These rules apply to conditional directives:

- One **!else** directive is allowed between **!if**, **!ifdef**, or **!ifndef** and **!endif** directives.
- Multiple **!elif** directives are allowed between **!if**, **!ifdef**, or **!ifndef**, **!else** and **!endif**.
- You can't split rules across conditional directives.
- You can nest conditional directives.
- **!if**, **!ifdef**, and **!ifndef** must have matching **!endif** directives within the same file.

The following information can be included between **!if** and **!endif** directives:

Macro definition	!include directive
Explicit rule	!error directive
Implicit rule	!undef directive

In an if statement, a conditional expression consists of decimal, octal, or hexadecimal constants and the operators shown in the following table.

Operator	Description	Operator	Description
-	Negation	?:	Conditional expression
~	Bit complement	!	Logical NOT
+	Addition	>>	Right shift
-	Subtraction	<<	Left shift
*	Multiplication	&	Bitwise AND
/	Division	|	Bitwise OR
%	Remainder	^	Bitwise XOR
&&	Logical AND	>=	Greater than or equal*
||	Logical OR	<=	Less than or equal*
>	Greater than	==	Equality*
<	Less than	!=	Inequality*

*Operator also works with string expressions.

MAKE evaluates a conditional expression as either a 32-bit signed integer or a character string.

!include

This directive is like the **#include** preprocessor directive for the C or C++ language-it lets you include the text of another file in the makefile:

```
!include filename
```

You can enclose filename in quotation marks (" ") or angle brackets (< >) and nest directives to unlimited depth, but writing duplicate **!include** directives in a makefile isn't permitted-you'll get the error message cycle in the include file.

Rules, commands, or directives must be complete within a single source file; you can't start a command in an **!include** file, then finish it in the makefile.

MAKE searches for **!include** files in the current directory unless you've specified another directory with the **-I** command-line option.

!message

The **!message** directive lets you send messages to the screen from a makefile. You can use these messages to help debug a makefile that isn't working properly. For example, if you're having trouble with a macro definition, you could put this line in your makefile:

```
!message The macro is defined here as: $(MacroName)
```

When MAKE interprets this line, it will print onscreen (assuming the macro expands to .CPP):

```
The macro is defined here as: .CPP
```

.path.ext

The **.path.ext** directive tells MAKE where to look for files with a certain extension. The following example tells MAKE to look for files with the .c extension in C:\SOURCE or C:\CFILES and to look for files with the .obj extension in C:\OBJS.

```
.path.c = C:\CSOURCE;C:\CFILES
.path.obj = C:\OBJS
```

.precious

If a MAKE build fails, MAKE deletes the target file. The **.precious** directive prevents the file deletion, which you might desire for certain kinds of targets. For example, if your build fails to add a module to a library, you might not want the library to be deleted.

The syntax for .precious is

```
.precious: target [target ...]
```

.suffixes

The **.suffixes** directive tells MAKE the order (by file extensions) for building implicit rules.

The syntax of **.suffixes** is

```
.suffixes: .ext [.ext ...]
```

where .ext represents the dependent file extensions you list in your implicit rules. For example, you could include the line `.suffixes: .asm .c .cpp` to tell MAKE to interpret implicit rules beginning with the ones dependent on .ASM files, then .C files, then .CPP files, regardless of what order they appear in the makefile.

The following **.suffixes** example tells MAKE to look for a source file first with an .ASM extension, next with a .C extension, and finally with a .CPP extension. If MAKE finds MYPROG.ASM, it builds MYPROG.OBJ from the assembler file by calling TASM. MAKE then calls TLINK; otherwise, MAKE searches for MYPROG.C to build the .OBJ file or it searches for MYPROG.CPP.

```
.suffixes: .asm .c .cpp

myprog.exe: myprog.obj
    bcc myprog.obj

.cpp.obj:
    bcc -P -c $<
.asm.obj:
    tasm /mx $<
.c.obj:
    bcc -P- -c $<
```

!undef

!undef (undefine) clears the given macro, causing an **!ifdef** *MacroName* test to fail.

The syntax of the **!undef** directive is

```
!undef MacroName
```

Using macros in directives

You can use the **$d** macro with the **!if** conditional directive to perform some processing if a specific macro is defined. Follow the **$d** with macro name enclosed in parentheses or braces, as shown in the following example:

```
!if $d(DEBUG)                 #If DEBUG is defined,
bcc -v f1.cpp f2.cpp          #compile with debug information;
!else                         #otherwise
bcc -v- f1.cpp f2.cpp         #don't include debug information.
!endif
```

Null macros

While an undefined macro name causes an **!ifdef** *MacroName* test to return false, *MacroName* defined as null will return true. You define a null macro by following the equal sign in the macro definition with either spaces or a return character. For example, the following line defines a null macro in a makefile:

```
NULLMACRO =
```

Either of the following lines can define a null macro on the MAKE command line:

```
NULLMACRO=""
-DNULLMACRO
```

TDUMP: The file dumping utility

TDUMP.EXE produces a file dump that shows the structure of a file.

TDUMP breaks apart a file structurally and uses the file's extension to determine the output display format. TDUMP recognizes many file formats, including .EXE, .OBJ, and .LIB files. If TDUMP doesn't recognize an extension, it produces a hexadecimal dump of the file. You can control the output format by using the TDUMP command-line options when you start the program.

TDUMP's ability to peek at a file's inner structure displays not only a file's contents, but also how a file is constructed. Moreover, because TDUMP verifies that a file's structure matches its extension, you can also use TDUMP to test file integrity.

TDUMP syntax

The command-line syntax for TDUMP is:

```
TDUMP [<options>] <Inputfile> [<Listfile>] [<options>]
```

`<Inputfile>` is the file whose structure you want to display (or "dump")

`<Listfile>` is an optional output file name (you can also use the standard DOS redirection command ">")

`<options>` stand for any of the TDUMP command-line options

Note: For more information on using TDUMP, refer to the online file TDUMP.TXT.

TDUMP command-line options

You can use several optional switches with TDUMP, all of which start with a hyphen or a forward slash. The following two examples are equivalent:

```
TDUMP -el -v demo.exe
TDUMP /el /v demo.exe
```

The -a and -a7 options

TDUMP automatically adjusts its output display according to the file type. You can force a file to be displayed as ASCII by including the **-a** or **-a7** option.

-a produces an ASCII file display, which shows the offset and the contents in displayable ASCII characters. A character that is not displayable (like a control character) appears as a period.

-**a7** converts high-ASCII characters to their low-ASCII equivalents. This is useful if the file you are dumping sets high-ASCII characters as flags (WordStar files do this).

The -b# option

The **-b#** option allows you to display information beginning at a specified offset. For example, if you wanted a dump of MYFILE starting from offset 100, you would use:

```
TDUMP -b100 MYFILE
```

The -d option

The **-d** option causes TDUMP to dump any Borland 32-bit debug information found in the .OBJ file. If you do not specify this option, TDUMP displays raw data only.

The -e, -el, -er and -ex options

All four options force TDUMP to display the file as an executable (.EXE) file.

An .EXE file display consists of information contained within a file that is used by the operating system when loading a file. If symbolic debugging information is present (Turbo Debugger or Microsoft CodeView), TDUMP displays it.

TDUMP displays information for DOS executable files, NEW style executable files (Microsoft Windows and OS/2 .EXEs and DLLs), Linear Executable files, and Portable Executable (PE) files used by Windows NT and Windows 95.

- **-el** suppresses line numbers in the display.
- **-er** prevents the relocation table from displaying.
- **-ex** prevents the display of New style executable information. This means TDUMP only displays information for the DOS "stub" program.

The -h option

The **-h** option displays the dump file in hexadecimal (hex) format. Hex format consists of a column of offset numbers, 16 columns of hex numbers, and their ASCII equivalents (a period appears where no displayable ASCII character occurs).

If TDUMP doesn't recognize the input file's extension, it displays the file in hex format (unless an option is used to indicate another format).

The -l and -li options

The **-l** option displays the output file in library (.LIB) file format. A library file is a collection of object files (see the **-o** option for more on object files). The library file dump displays library-specific information, object files, and records in the object file.

The **-li** option tells TDUMP to display a short form of "impdef" records when dumping import libraries. You can also specify a search string using the following syntax:

```
-li=<string>
```

For example, the command

```
TDUMP -li=codeptr import.lib
```

results in the following output:

```
Impdef:(ord) KERNAL.0336=ISBADCODEPTR
```

This output shows that the function is exported by ordinal, whose ordinal value is 336 (decimal). In addition, the output displays the module and function name.

If you give the command

```
TDUMP -li=walk import32.lib
```

TDUMP displays:

```
Impdef:(name) KERNEL32.????=HEAPWALK
```

This shows the output of a function exported by name.

The -m option

The **-m** option leaves C++ names occurring in object files, executable files, and Turbo Debugger symbolic information files in "mangled" format. This option is helpful in determining how the C++ compiler "mangles" a given function name and its arguments.

The -o, -oc, -ox, and -oi options

-o displays the file as an object (.OBJ) file. An object file display contains descriptions of the command records that pass commands and data to the linker, telling it how to create an .EXE file. The display format shows each record and its associated data on a record-by-record basis.

-oc causes TDUMP to perform a cyclic redundancy test (CRC) on each encountered record. The display differs from the -o display only if an erroneous CRC check is encountered (the TDUMP CRC value differs from the record's CRC byte).

-ox<id> excludes designated record types from the object module dump. Replace <id> with the record name not to be displayed. For instance,

```
TDUMP -oxPUBDEF MYMODULE.OBJ
```

produces an object module display for MYMODULE.OBJ that excludes the PUBDEF records.

-oi<id> includes only specified record types in the object module dump. Replace <id> with the name of the record to be displayed. For instance,

```
TDUMP -oiPUBDEF MYMODULE.OBJ
```

produces an object module display for MYMODULE.OBJ that displays only the PUBDEF records.

The **-ox** and **-oi** options are helpful in finding errors that occur during linking. By examining the spelling and case of the EXTDEF symbol and the PUBDEF symbol, you can resolve many linking problems. For instance, if you receive an "unresolved external" message from the linker, use `TDUMP -oiEXTDEF` to display the external definitions occurring in the module causing the error. Then, use `TDUMP -oiPUBDEF` on the module containing the public symbol the linker could not match.

Another use for the **-oi** switch is to check the names and sizes of the segments generated in a particular module. For instance,

```
TDUMP -oiSEGDEF MYMODULE.OBJ
```

displays the names, attributes, and sizes of all of the segments in MYMODULE.

Note: To get a list of record types for **-oi** and **-ox**, use the command-line options **-oi?** and **-ox?**.

The -R option

The **-R** option causes TDUMP to dump relocation tables from 32-bit PE (Win32) format images. The default is to suppress these dumps.

The -v option

The **-v** option is used for verbose display. If used with an .OBJ or .LIB file, TDUMP produces a hexadecimal dump of the record's contents without any comments about the records.

If you use TDUMP on a Turbo Debugger symbol table, it displays the information tables in the order in

which it encounters them. TDUMP doesn't combine information from several tables to give a more meaningful display on a per-module basis.

Using TLINK32 and ILINK32

See also

TLINK32 and ILINK32 are the command-line tools that combine object modules (.OBJ files), library modules (.LIB files), and resources to produce executable files (.EXE and .DLL files). Because the compiler automatically calls a linker, you don't need to explicitly use a linker unless you suppress the linking stage of compiling (see the **-c** compiler option).

TLINK32 and ILINK32 are invoked from the command line to link a configuration file called TLINK32.CFG (ILINK32 uses ILINK32.CFG), an optional response file, and command-line options to link object modules, libraries, and resources into an executable file.

TLINK32 and ILINK32 link 32-bit Windows code and use the resource linker RLINK32.DLL.

TLINK32 command-line syntax

The linker syntax controls how the linkers work. Linker command-line options are case-sensitive. Unless specified, instructions and options for TLINK32 also apply to ILINK32.

The linkers can also use a configuration file called TLINK32.CFG for options that you'd normally type at the command-line.

Syntax

```
TLINK32 | iLINK [@respfile][options] startup myobjs, [exe], [mapfile],
  [libraries], [deffile], [resfile]
```

`[@respfile]`	A response file is an ASCII file that lists linker options and file names that you would normally type at the command line. By placing options and files names in a response file, you can save the amount of keystrokes you need to type to link your application.
`[options]`	Linker options that control how the linker works. For example, options specify whether to produce an .EXE or a DLL file. Linker options must be preceded by either a slash (/) or a hyphen (-).
`startup`	A Borland initialization module for executables or DLLs that arranges the order of the various segments of the program. Failure to link in the correct initialization module usually results in a long list of error messages telling you that certain identifiers are unresolved, or that no stack has been created.
`myobjs`	The .OBJ files you want linked. Specify the path if the files arn't in the current directory. (The linker appends an .OBJ extensions if no extension is present.)
`[exe]`	The name you want given to the executable file (.COM, .EXE, or .DLL). If you don't specify an executable file name, TLINK derives the name of the executable by appending .EXE or .DLL to the first object file name listed. (The linker assumes or appends an .EXE extensions for executable files if no extension is present. It also assumes or appends a .DLL extention for dynamic link libraries if no extension is present.)
`[mapfile]`	Is the name you want given to the map file. If you don't specify a name, the map file name is given the same as exefile (but with the .MAP extension). (The linker appends a .MAP extensions if no extension is present.)
`[libraries]`	The library files you want included at link time. Do not use commas to separate the libraties listed. If a file is not in the current directory or the search path (see the **/L** option) then you must include the path in the link statement. (The linker appends a .LIB extension if no extension is present.) The order in which you list the libraries is very important; be sure to use the order defined in this list: ▪ Code Guard libraries (if needed) ▪ List any of your own user libraries, noting that if a function is defined more than once, the linker uses the first definition encountered ▪ If you’re creating a DOS overlay, link the DOS overlay module OVERLAY.LIB ▪ DPMI libraries (DOS DPMI applications only) ▪ IMPORT.LIB (if you’re creating an executable that uses the Windows API) ▪ Math libraries (if needed) ▪ Runtime libraries associated with your memory model and platform

`[deffile]` The module definition file for a Windows executable. If you don't specify a module definition (.DEF) file and you have used the **/Twe** or **/Twd** option, the linker creates an application based on default settings. (The linker appends a .DEF extension if no extension is present.)
`[resfile]` A list of.RES files (compiled resource files) to bind to the executable. (The linker appends an .RES extension if no extension is present.)

TLINK32.CFG File

TLINK32 uses a configuration file called TLINK32.CFG for options that you would normally type at the command line (note that configuration files can contain only options, not file names). Configuration files let you save options you use frequently, so you do not have to continually retype them.

TLINK32 looks for TLINK32.CFG in the current directory, then in the directory from which it was loaded.

The following TLINK32.CFG file tells TLINK32 to look for libraries first in the directory C:\BORLANDC\LIB then in C:\WINAPPS\LIB, to include debug information in the executables it creates, to create a detailed segment map, and to produce a Windows executable (.EXE not .DLL).

```
;Sample TLINK32.CFG file
/Lc:\BORLANDC\LIB;c:\WINAPPS\LIB
/v /s
/Tpe
```

Note: If you specify command-line options in addition to those recorded in a configuration file, the command-line options override any conflicting configuration options.

Linker response files

You can use response files with the command-line linkers to specify linker options.

Response files are ASCII files that list linker options and file names that you would normally type at the command line. Response files allow you longer command lines than most operating systems support, plus you don't have to continually type the same information. Response files can include the same information as configuration files, but they also support the inclusion of file names.

Unlike the command line, a response file can be several lines long. To specify an added line, end a line with a plus character (+) and continue the command on the next line. Note that if a line ends with an option that uses the plus to turn it on (such as **/v+**), the + is not treated as a line continuation character (to continue the line, use **/v+ +**).

If you separate command-line components (such as .OBJ files from .LIB files) by lines in a response file, you must leave out the comma used to separate them on the command line. For example,

```
/c c0ws+
myprog,myexe +
mymap +
mylib cws
```

leaves out the commas you'd have to type if you put the information on the command line:

```
TLINK32 /c c0ws myprog,myexe,mymap,mylib cws
```

To use response files,

1. Type the command-line options and file names into an ASCII text file and save the file. Response files shipped with Ebony have an .RSP extension.
2. Type `TLINK32 @[path]RESFILE.RSP` where `RESFILE.RSP` is the name of your response file.

You can specify more than one response file as follows:

```
tlink32 /c @listobjs.rsp,myexe,mymap,@listlibs.rsp
```

Note: You can add comments to response files using semicolons; the linker ignores any text on a line that follows a semicolon.

Using TLINK32 with BCC32.EXE

You can pass options and files to TLINK through the command-line compilers (BCC.EXE and BCC32.EXE) by typing file names on the command line with explicit .OBJ and .LIB extensions. For example,

```
BCC mainfile.obj sub1.obj mylib.lib
```

links MAINFILE.OBJ, SUB1.OBJ, and MYLIB.LIB tp produce the executable MAINFILE.EXE.

Note: By default, BCC starts TLINK with the files C0WS.OBJ, CWS.LIB, and IMPORT.LIB (initialization module, run-time library, and Windows import library). BCC32 starts TLINK32 with the files C0W32.OBJ, CW32.LIB, and IMPORT32.LIB. In addition, both compilers always pass the linkers the **/c** (case-sensitive link) option.

Module definition files

Example

The module definition file is an ASCII text file that provides information to TLINK32 about the contents and system requirements of a Windows application. You can create a module definition file using IMPDEF, and you can create import libraries from module definition files using IMPLIB.

The module definition file names the .EXE or .DLL, identifies the application type, lists imported and exported functions, describes the code and data segment attributes, and lets you specify attributes for additional code and data segments, specifies the size of the stack, and provides for the inclusion of a stub program.

Specific elements of this module definition file are:

- CODE statement
- DATA statement
- DESCRIPTION statement
- EXETYPE statement
- EXPORTS statement
- HEAPSIZE statement
- IMPORTS statement
- LIBRARY statement
- NAME statement
- SECTIONS statement
- SEGMENTS statement
- STACKSIZE statement
- STUB statement
- SUBSYSTEM statement

When a module definition file is not specified

If no module definition file is specified, the following defaults are assumed:

```
CODE        PRELOAD MOVEABLE DISCARDABLE
DATA        PRELOAD MOVEABLE MULTIPLE   ; (for applications)
            PRELOAD MOVEABLE SINGLE   ; (for DLLs)
HEAPSIZE    4096
STACKSIZE   1048576
```

To change an applications attributes from these defaults, you will need to create a module definition file.

If you delete the EXETYPE statement, the C++Builder linker can determine what kind of executable you want to produce from your settings in the IDE or the options you supply on the command line.

You can include an import library to substitute for the IMPORTS section of the module definition.

You can use the **_export** keyword in the definitions of export functions in your C and C++ source code to remove the need for an EXPORTS section. Note, however, that if **_export** is used to export a function, that function is exported by name rather than by ordinal (ordinal is usually more efficient).

CODE statement (Module Definition File)

CODE defines the default attributes of code segments. Code segments can have any name, but must belong to segment classes whose name ends in CODE (such as CODE or MYCODE). The 32-bit syntax is:

```
CODE [PRELOAD | LOADONCALL]
     [EXECUTEONLY | EXECUTEREAD]
```

- PRELOAD means code is loaded when the calling program is loaded.
- LOADONCALL (the default) means the code is loaded when called by the program.
- EXECUTEONLY means a code segment can only be executed.
- EXECUTEREAD (the default) means the code segment can be read and executed.
- FIXED (the default) means the segment remains at a fixed memory location.
- MOVEABLE means the segment can be moved.
- DISCARDABLE means the segment can be dicsarded if it is no longer needed (this implies MOVEABLE).
- NONDISCARDABLE (the default) means the segment can not be discarded.

DATA statement (Module Definition File)

DATA defines attributes of data segments. The syntax is:

```
DATA [NONE | SINGLE | MULTIPLE]
     [READONLY | READWRITE]
     [PRELOAD | LOADONCALL]
     [SHARED | NONSHARED]
```

- NONE means that there is no data segment created. This option is available only for libraries.
- SINGLE (the default for .DLLs) means a single data segment is created and shared by all processes.
- MULTIPLE (the default for .EXEs) means that a data segment is created for each process.
- READONLY means the data segment can be read only.
- READWRITE (the default) means the data segment can be read and written to.
- PRELOAD means the data segment is loaded when a module that uses it is first loaded.
- LOADONCALL (the default) means the data segment is loaded when it is first accessed (LOADONCALL is ignored for 32-bit applications).
- SHARED (the default for 16-bit .DLLs) means one copy of the data segment is shared among all processes.
- NONSHARED (the default for programs and 32-bit .DLLs) means a copy of the data segment is loaded for each process needing to use the data segment.

DESCRIPTION statement (Module Definition File)

DESCRIPTION (optional) inserts text into the application module and is typically used to embed author, date, or copyright information. The syntax is:

```
DESCRIPTION 'Text'
```

`Text` is an ASCII string delimited with single quotes.

EXETYPE statement (Module Definition File)

EXETYPE defines the default executable file (.EXE) header type for 16-bit applications. You can leave this section in for 32-bit applications for backward compatibility, but if you need to change the EXETYPE, see the NAME statement. The syntax for EXETYPE is:

```
EXETYPE  [WINDOWAPI] | [WINDOWCOMPAT] | [NOTWINDOWCOMPAT]
```

- WINDOWAPI is a Windows executable, and is equivalent to the TLINK option **/aa**.
- WINDOWCOMPAT is a Windows-compatible character-mode executable, and is equivalent to the TLINK option **/ap**.
- NOTWINDOWCOMPAT is a character-mode application which won't run under Windows. It is equivalent to the TLINK option **/ai**.

EXPORTS statement (Module Definition File)

EXPORTS defines the names and attributes of functions to be exported. The EXPORTS keyword marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line. The syntax is:

```
EXPORTS
   ExportName [Ordinal]
   [RESIDENTNAME] [Parameter]
```

- ExportName specifies an ASCII string that defines the symbol to be exported as follows:
  `EntryName [=InternalName]`
  `InternalName` is the name used within the application to refer to this entry.
  `EntryName` is the name listed in the executable file's entry table and is externally visible.
- `Ordinal` defines the function's ordinal value as follows:

  `@ordinal`

  where `ordinal` is an integer value that specifies the function's ordinal value.

  When an application or DLL module calls a function exported from a DLL, the calling module can refer to the function by name or by ordinal value. It's faster to refer to the function by ordinal because string comparisons aren't required to locate the function. To use less memory, export a function by ordinal (from the point of view of that function's DLL) and import/call a function by ordinal (from the point of view of the calling module).

  When a function is exported by ordinal, the name resides in the nonresident name table. When a function is exported by name, the name resides in the resident name table. The resident name table for a module is in memory whenever the module is loaded; the nonresident name table isn't.
- RESIDENTNAME specifies that the function's name must be resident at all times. This is useful only when exporting by ordinal (when the name wouldn't be resident by default).
- `Parameter` is an optional integer value that specifies the number of words the function expects to be passed as parameters.

HEAPSIZE statement (Module Definition File)

HEAPSIZE defines the number of bytes the application needs for its local heap. An application uses the local heap whenever it allocates local memory. The support for HEAPSIZE is slightly different for 16-bit or 32-bit applications.

The 16-bit syntax for HEAPSIZE is:

```
HEAPSIZE Allocate
```

- `Allocate` is an integer value which specifies the amount of heap allocated at program startup. For 16-bit applications, this size cannot exceed the physical segment size of 65,535 bytes (64K).

The 32-bit syntax for HEAPSIZE is:

```
HEAPSIZE Reserve[, Commit]
```

- `Reserve` can be a decimal or hex value, the default of which is 1MB. To help with backward (16-bit) compatibility, the linker uses the default value of 1MB if you specify in the .DEF file a reserve value less than 64K.
- `Commit` is a decimal or hex value. The commit size is optional, and if not specified defaults to 4K. The minimum commit size you can specify is 0. In addition, the specified or default commit size must always be smaller or equal to the reserve size.

Reserved memory refers to the maximum amount of memory that can be allocated either in physical memory or in the paging file. In other words, reserved memory specifies the maximum possible heap size. The operating system guarantees that the specified amount of memory will be reserved and, if necessary, allocated.

The meaning of committed memory varies among operating systems. In Windows NT, commited memory refers to the amount of physical memory allocated for the heap at application load/initialization time. Committed memory causes space to be allocated either in physical memory or in the paging file. A higher commit value saves time when the application needs more heap space, but increases memory requirements and possible startup time.

You can override any heap reserve or commit size specified in the .DEF file with the **/H** or **/Hc** command-line options. **/H** lets you specify a heap reserve size less than the 64K minimum allowed in the .DEF file.

IMPORTS statement (Module Definition File)

IMPORTS defines the names and attributes of functions to be imported from DLLs. Instead of listing imported DLL functions in the IMPORTS statement, you can either

- specify an import library for the DLL in the TLINK command line, or
- include the import library for the DLL in the project manager in the IDE.

If you are programming for 32 bits, you must use **_ _import** to import any function, class, or data you want imported. For 16 bits, you must use **_ _import** with the classes you want imported.

The IMPORTS keyword marks the beginning of the definitions followed by any number of import definitions, each on a separate line. The syntax is:

```
IMPORTS
[InternalName=]ModuleName.Entry
```

- `InternalName` is an ASCII string that specifies the unique name the application uses to call the function.
- `ModuleName` specifies one or more uppercase ASCII characters that define the name of the executable module containing the function. The module name must match the name of the executable file. For example, the file SAMPLE.DLL has the module name SAMPLE.
- `Entry` specifies the function to be imported--either an ASCII string that names the function or an integer that gives the function's ordinal value.

LIBRARY statement (Module Definition File)

LIBRARY defines the name of a DLL module. A module definition file can contain either a LIBRARY statement to indicate a .DLL or a NAME statement to indicate a .EXE.

A library's module name must match the name of the executable file. For example, the library MYLIB.DLL has the module name MYLIB. The syntax is:

```
LIBRARY LibraryName [INITGLOBAL | INITINSTANCE]
```

- `LibraryName` (optional) is an ASCII string that defines the name of the library module. If you don't include a `LibraryName`, TLINK uses the file name with the extension removed. If the module definition file includes neither a NAME nor a LIBRARY statement, TLINK assumes a NAME statement without a `ModuleName` parameter.
- INITGLOBAL means the library-initialization routine is called only when the library module is first loaded into memory.
- INITINSTANCE means the library-initialization routine is called each time a new process uses the library.

NAME statement (Module Definition File)

NAME is the name of the application's executable module. The module name identifies the module when exporting functions. For 32-bit applications, NAME must appear before EXETYPE. If NAME and EXETYPE don’t specify the same target type, the linker uses the type listed with NAME. The syntax is:

```
NAME ModuleName [WINDOWSAPI] | [WINDOWCOMPAT]
```

- `ModuleName` (optional) specifies one or more uppercase ASCII characters that name the executable module. The name must match the name of the executable file. For example, an application with the executable file SAMPLE.EXE has the module name SAMPLE.

  If `ModuleName` is missing, TLINK assumes that the module name matches the file name of the executable file. For example, if you do not specify a module name and the executable file is named MYAPP.EXE, TLINK assumes that the module name is MYAPP.

  If the module definition file includes neither a NAME nor a LIBRARY statement, TLINK assumes a NAME statement without a `ModuleName` parameter.

- WINDOWSAPI specifies a Windows executable, and is equivalent to the TLINK32 option **/aa**.
- WINDOWCOMPAT spedifies a Windows-compatible character-mode executable, and is equivalent to the TLINK32 option **/ap**.

SECTIONS statement (Module Definition File)

The SECTIONS statement lets you set attributes for one or more section in the image file. you can use this You can use this statement to override the default attributes for each different type of section. The syntax for SECTIONS is:

```
SECTIONS
  <section_name> (CLASS 'classname'] attributes
```

- SECTIONS marks the beginning of a list of section definitions.
- After the SECTIONS keyword, each section definition must be listed on a separate line. Note that the SECTIONS keyword can be on the same line as the first definition or on a preceding line. In addition, the .DEF file can contain one or more SECTIONS statements. The SEGMENTS keyword is supported as a synonym for SECTIONS. The syntax for the individual section listings is as follows:

In this syntax, section_name is case sensitive. The CLASS keyword is supported for compatibility but is ignored. The attributes argument can be one or more of the following: EXECUTE, READ, SHARED, and WRITE.

SEGMENTS statement (Module Definition File)

SEGMENTS defines the segment attributes of additional code and data segments. The syntax is:

```
SEGMENTS
   SegmentName [CLASS 'ClassName']
   [MinAlloc]
   [SHARED | NONSHARED]
   [PRELOAD | LOADONCALL]
   [MIXED1632]
```

- `SegmentName` is a character string that names the new segment. It can be any name, including the standard segment names _TEXT and _DATA, which represent the standard code and data segments.
- `ClassName` (optional) is the class name of the specified segment. If no class name is specified, TLINK uses the class name CODE.
- `MinAlloc` (optional) is an integer that specifies the minimum allocation size for the segment. TLINK and TLINK32 ignore this value.
- SHARED (the default for 16-bit .DLLs) means one copy of the segment is shared among all processes.
- NONSHARED (the default for .EXEs and 32-bit .DLLs) means a copy of the segment is loaded for each process needing to use the data segment.
- PRELOAD means that the segment is loaded immediately.
- LOADONCALL means that the segment is loaded when it is accessed or called (this is ignored by TLINK32). The Resource Compiler may override the LOADONCALL option and preload segments instead.
- MIXED1632 (optional) is supported by the 16-bit linker only, and lets you link 32-bit modules with your 16-bit Windows 95 applications. The Windows 95 16-bit loader supports 32-bit segments when the 2000H bit is set in the segment table of the application.

STACKSIZE statement (Module Definition File)

STACKSIZE defines the number of bytes the application needs for its local stack. An application uses the local stack whenever it makes function calls. The support for STACKSIZE is slightly different for 16-bit or 32-bit applications.

The 16-bit syntax for STACKSIZE is:

```
STACKSIZE Allocate
```

- `Allocate` is an integer value which specifies the amount of stack allocated at program startup. For 16-bit applications, this size cannot exceed the physical segment size of 65,535 bytes (64K).

The 32-bit syntax for STACKSIZE is:

```
STACKSIZE Reserve[, Commit]
```

- `Reserve` can be a decimal or hex value, the default of which is 1MB. To help with backward (16-bit) compatibility, the linker uses the default value of 1MB if you specify in the .DEF file a reserve value less than 64K.
- `Commit` is a decimal or hex value. The commit size is optional, and if not specified defaults to 8K. The minimum commit size you can specify is 4K. In addition, the specified or default commit size must always be smaller or equal to the reserve size.

Reserved memory refers to the maximum amount of memory that can be allocated either in physical memory or in the paging file. In other words, reserved memory specifies the maximum possible stack size. The operating system guarantees that the specified amount of memory will be reserved and, if necessary, allocated.

The meaning of committed memory varies among operating systems. In Windows NT, commited memory refers to the amount of physical memory allocated for the stack at application load/initialization time. Committed memory causes space to be allocated either in physical memory or in the paging file. A higher commit value saves time when the application needs more stack space, but increases memory requirements and possible startup time.

You can override any stack reserve or commit size specified in the .DEF file with the **/S** or **/Sc** command-line options. **/S** lets you specify a stack reserve size less than the 64K minimum allowed in the .DEF file.

Note: Do not use the STACKSIZE statement when compiling .DLLs.

STUB statement (Module Definition File)

STUB appends a DOS executable file specified by `FileName` to the beginning of the module. The executable stub displays a warning message and terminates if the user attempts to run the executable stub in the wrong environment (running a Windows application under DOS, for example).

Borland C++ adds a built-in stub to the beginning of a Windows application unless a different stub is specified with the STUB statement. You should not use the STUB statement to include WINSTUB.EXE because the linker does this automatically.

The syntax is:

```
STUB 'FileName'
```

- `FileName` is the name of the DOS executable file to be appended to the module. The name must have the DOS file name format. If the file named by `FileName` is not in the current directory, TLINK searches for the file in the directories specified by the PATH environment variable.

SUBSYSTEM statement (Module Definition File)

SUBSYSTEM lets you specify the Windows subsystem and subsystem version number for the application being linked. The syntax for SUBSYSTEM is:

```
SUBSYSTEM [subsystem,]subsystemID
```

- The optional parameter `subsystem` can be any one of the following values: WINDOWS, WINDOWAPI, WINDOWCOMPAT, NOTWINDOWCOMPAT. If you do not specify a `subsystem`, the linker defaults to a WINDOWS subsystem.
- You must specify the `subsystemID` parameter using the format `d.d` where `d` is a decimal number. For example, if you want to specify Windows 4.0, you could use either one of the following two SUBSYSTEM statements:

```
SUBSYSTEM 4.0
SUBSYSTEM WINDOWS,4.0
```

You can override any SUBSYSTEM statement in a .DEF file using the **/a** and **/V** command-line options.

Example module definition file

Here's a module definition file to serve as an example.

```
NAME            WHELLO
DESCRIPTION     'C++ Windows Hello World'
EXETYPE         WINDOWS
CODE            MOVEABLE
DATA            MOVEABLE MULTIPLE
HEAPSIZE        1024
STACKSIZE       5120
EXPORTS         MainWindowProc
```

Let's take this file apart, statement by statement:

- NAME specifies a name for an application. If you want to build a .DLL instead of an application, you would use LIBRARY instead of NAME. Every module definition file should have either a NAME or a LIBRARY statement, but never both. The name specified must be the same name as the executable file.
- DESCRIPTION lets you specify a string that describes your application or library.
- EXETYPE can be either WINDOWS or OS2. Only WINDOWS is supported in this version of Borland C++.
- CODE defines the default attributes of code segments. The MOVEABLE option means that the code segment can be moved in memory at run time.
- DATA defines the default attributes of data segments. MOVEABLE means that it can be moved in memory at run time. Windows lets you run more than one instance of an application at the same time. In support of that, the MULTIPLE options ensures that each instance of the application has its own data segment.
- HEAPSIZE specifies the size of the application's local heap.
- STACKSIZE specifies the size of the application's local stack. You can't use the STACKSIZE statement to create a stack for a .DLL.
- EXPORTS lists those functions in the WHELLO application that can be called by other applications or by Windows. Functions that are intended to be called by other modules are called callbacks, callback functions, or export functions.
- To help you avoid the necessity of creating and maintaining long EXPORTS sections, Borland C++ provides the _export keyword. Functions flagged with **_export** are identified by the linker and entered into an export table for the module. If the Smart Callbacks option is used at compile time (**/WS** on the BCC command-line, or Options|Compiler|Entry/Exit Code|Windows Smart Callbacks), then callback functions do not need to be listed either in the EXPORTS statement or flagged with the **_export** keyword. Borland C++ compiles them in such a way so that they can be callback functions.
- This application doesn't have an IMPORTS statement, because the only functions it calls from other modules are those from the Windows API; those functions are imported via the automatic inclusion of the IMPORT.LIB import library. When an application needs to call other external functions, these functions must be listed in the IMPORTS statement, or included via an import library.
- This application doesn't include a STUB statement. Borland C++ uses a built-in stub for Windows applications. The built-in stub simply checks to see if the application was loaded under Windows, and, if not, terminates the application with a message that Windows is required. If you want to write and include a custom stub, specify the name of that stub with the STUB statement.

TRIGRAPH: A Character-Conversion Utility

Trigraphs are three-character sequences that replace certain characters used in the C language that are not available on some keyboards. Translating trigraphs in the compiler would slow compilation down considerably, so Borland C++ provides a filter named TRIGRAPH.EXE to handle trigraph sequences when you need to. The syntax for invoking this program is as follows:

```
TRIGRAPH [-u] file(s) [...]
```

The following table shows the trigraph sequences that TRIGRAPH.EXE recognizes:

Trigraph	Characters
??=	#
??(	[
??)	]
??/	\
??'	^
??<	{
??>	}
??!	|
??-	~