

TopSpeed®

For IBM® Personal Computers and Compatibles

Developer's Guide

TopSpeed Corporation

Copyright© 1989-1991, by TopSpeed Corporation. All rights reserved.

TopSpeed® is a registered trademark of TopSpeed Corporation.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Printed in the United Kingdom.

Part number 01-01010-3000-0102

10 9 8 7 6 5 4 3 2 1

Contents

CHAPTER 1	7
Introduction	7
CHAPTER 2	9
The Project System	9
Overview	9
Smart Linking	9
SmartMethod Linking	10
Constant Sharing	10
Type-safe Linking	10
A Simple Project File	11
Miscellaneous Commands	26
Predefined Compilers	30
Memory Models	31
Linking	32
Special Project Macros	32
CHAPTER 3	35
TopSpeed from the Command Line	35
Command Line Overview	35
Compile Mode	35
Make Mode	36
Link Mode	36
Command-line Options	37
TSC Options	37
Compilation and Project Options	38
CHAPTER 4	42
Pragmas	42
Conventions and Syntax	42
Modula-2 Pragma Syntax	43
Pascal Pragma Syntax	44

C and C++ Pragma Syntax	44
Project System Pragma Syntax	44
Pragmas	44
Call Pragmas	45
Data Pragmas	51
Check Pragmas	56
The expr Pragma	57
Name Pragmas	57
Optimize Pragmas	58
Module Pragmas	61
Option Pragmas	63
Warn Pragmas	65
The project Pragma	70
The save and restore Pragmas	70
Link Pragmas	71
Link_option Pragmas	71
The define Pragma	73
CHAPTER 5	74
Conditional Compilation	74
Modula-2 Syntax	74
Pascal Syntax	75
C and C++ Syntax	76
Predefined Identifiers	76
CHAPTER 6	78
Segment-based Overlays	78
Overview	78
Compiling and Running an Overlayed Program	79
Design Considerations	79
Programming for the Overlay Model	82
Overlay System API	82
Run-time Errors	83
Limits and System Requirements	86
System Requirements	86

CHAPTER 7	87
Dynamic Link Libraries for OS/2	87
The Advantages of Dynamic Linking	87
Pitfalls of Dynamic Linking and their Solution	87
Understanding Dynamic Linking	88
Dynamic Linking	91
Creating Programs that use DLLs	95
Running Programs that use DLLs	95
An Example	95
Initialization in a DLL	98
Using INITDLL	98
Rules and Limitations for DLL Programs	99
Dynamic Link Loader Error Messages	100
CHAPTER 8	101
Multi-language Programming	101
Standard Cross Definition Files	101
Creating your own Cross Definition Files	102
Linking with your own Libraries	102
Type Equivalents	103
Strings	104
Enumeration Types	105
Calling Conventions	105
Naming Conventions	106
Library Considerations	107
Modula-2 and Pascal Initialization from C	109
Multi-language Object-Oriented Programming	110
Naming of Methods	111
Programming Examples	112
CHAPTER 9	113
Interfacing to Third-Party Code	113
Interfacing to Existing Code	113
Using Stack Frame Libraries	114
Assembly Language Interface	114

CHAPTER 10	115
Presentation Manager Programming	115
Program Source	115
Making PM Programs	115
Using a Resource Compiler	116
CHAPTER 11	117
Using CodeView and Compatible Debuggers	117
Using VID2CV	117
Limitations	117
CHAPTER 12	118
Miscellaneous Programming Considerations	118
Extending File Handle Limits	118
OS/2 Multi-thread Programming	118
APPENDIX A	119
Memory Models	119
The 80x86 Architecture — a Design Compromise	119
Assembly Language Interface	120
TopSpeed Language Extensions for Memory Models	120
The 8086 Architecture	120
The Standard Memory Models	121
Selecting Memory Models	128
Functions	130
APPENDIX B	137
Class Object Formats	137
Conventions	137
Pascal/Modula-2	137
Virtual Base Class Layout	142
Index	143

CHAPTER 1

Introduction

This guide describes the details of the TopSpeed Development System. The TopSpeed Development System provides the professional software developer with a large range of options for producing application programs, with all the necessary tools and libraries, using any of the TopSpeed programming languages.

The features of the standard TopSpeed Development System are described in this manual. Other, more specialized, features are described in the “*TopSpeed Advanced Programmer's Guide*”, part of the TopSpeed Techkit[®].

The Structure of this Guide

This manual is structured into the following chapters and appendices:

<i>Chapter 1</i>	provides an introduction to this guide.
<i>Chapter 2</i>	describes the TopSpeed Project System, a powerful system for controlling the building of programs.
<i>Chapter 3</i>	describes the use of the TopSpeed System from the command line, including a description of the command-line switches which are used to specify compilation options.
<i>Chapter 4</i>	describes TopSpeed's pragmas - a language-independent system of specifying compilation, project and linker options.
<i>Chapter 5</i>	describes how conditional compilation may be used in TopSpeed languages, and the symbols that are pre-defined by the Project System for this purpose.
<i>Chapter 6</i>	describes the powerful segment-based overlay system which TopSpeed programs can use to create executable programs up to 16 Mb in size.
<i>Chapter 7</i>	describes how OS/2 developers can make use of dynamic link libraries.
<i>Chapter 8</i>	describes how to interface between TopSpeed languages.
<i>Chapter 9</i>	describes a set of guidelines for developers who wish to interface to libraries not supplied by Jensen and Partners.
<i>Chapter 10</i>	describes special considerations when developing programs for OS/2's Presentation Manager.
<i>Chapter 11</i>	describes how to use TopSpeed with Microsoft CodeView and compatible debuggers.

<i>Chapter 12</i>	describes other, miscellaneous special programming considerations.
<i>Appendix A</i>	describes the predefined TopSpeed memory models, and how to create custom memory models to match your exact requirements. Special considerations for mixed memory-model programming are also described.
<i>Appendix B</i>	describes the format used by TopSpeed languages for storing objects with class type. This is of interest to developers using object-oriented features in a mixed language program, or who are interfacing an object-oriented program to assembler or third-party code.

Typographic Conventions

The following typographic conventions are used throughout this manual:

<i>Italics</i>	are used for emphasis and to introduce new concepts
Typewriter	is used for program examples and other items which would appear on the screen or in a program
Boldface	is used to highlight key letters in menus, such as P in Project
Keyboard	is used to show keys (like F1 or Enter) which you press at the keyboard

When it is necessary to show a combination of keys the following convention is used:

Alt-C

This means hold down the Alt key and press C. When a sequence of keys is combined with Alt or Ctrl, the following convention used:

Ctrl-K D

This convention means:

- Hold down the Ctrl key and press the K key.
- Release the Ctrl key and the K key.
- Press the D key.

CHAPTER 2

The Project System

Overview

The TopSpeed Project System, which is integrated into the TopSpeed Environment, is a powerful sequential language which combines the functionality of a batch processor, a linker and an intelligent *Make* system.

For simple projects, a small and easily read project file can be used. These files are plain ASCII text, and can be edited in the usual way. Alternatively, the TopSpeed Environment Project menu (see the “*TopSpeed User's Guide*”) can be used to create and edit project files automatically.

For more complicated projects, the powerful Project System language provides a high degree of control through the use of features such as macro substitution and conditional inclusion with complex conditional expressions.

For still more sophisticated users, much of the built-in logic of the Project System itself can be altered, allowing you to add new compilers or memory models, or to customize the behavior in other ways.

Unlike many project-control programs based on the original Unix make program, the TopSpeed Project System uses exact file dates to determine whether a target file is up to date with respect to a given source file, rather than simply checking that the target has a more recent timestamp than the source. For example, the TopSpeed Project System will correctly detect that a recompile is required when a source file is restored from a backup of an earlier version. In addition, the TopSpeed Project System takes into account the options with which a file was compiled, when checking that an object file is up to date. Thus, when a pragma is changed in a project file, the source files affected are automatically recompiled. The TopSpeed Project System determines source file dependencies automatically, from information stored in the object file, rather than requiring you to maintain dependency information manually.

Smart Linking

The TopSpeed Project System's linker automatically excludes any functions or data that are not required when building an executable file. This process is known as *smart linking*. This allows you to place as many related functions as you wish into the same source file without any overhead.

Smart linking is most useful when building large, general purpose libraries, such as the standard library. Without smart linking, for example, a library implementor must split various functions in the library into many different files, because it is unlikely that any one program uses every function in that library. These files make the overall design unclear and unstructured.

Smart linking allows you to place as many related functions as you like into the same file. The linker will ensure that only functions which are referenced are included in the final code. A good example of this is the library code for TopSpeed C, which contains less than 20 files.

SmartMethod Linking

The TopSpeed linker is also able to automatically exclude virtual methods that can never be called during the execution of a program. This feature allows the implementor of an object-oriented library to implement fully general library classes, without worrying about the overhead in the executable file. Any methods in a class that are never referenced will be excluded from the executable file, and only those that have actually been used, or which the linker cannot determine whether they have been used or not, will be included.

Constant Sharing

The TopSpeed linker is able to combine identical constants and code segments in an executable file, resulting in reductions in executable file size. In particular, virtual-method tables are automatically commoned up.

Type-safe Linking

Type-safe linking can save you hours of debugging and can catch bugs that you might otherwise overlook. Broadly speaking, it ensures that when you call a function which has been defined elsewhere, the parameters you pass to it correspond exactly to the parameters that the function expects.

Modula-2 and Pascal impose this constraint rigidly at compile time, but C does not. Type-safe linking is therefore more important for C applications. However, even in a Modula-2 program it becomes important if calls are made to externally-written procedures.

This can be illustrated with with a C example. Suppose you use an external function that has no prototype, and inadvertently call the function with the wrong parameter types. In traditional C systems, your program might crash, but with TopSpeed, the type-safe linker catches the error.

For example, one file might contain:

```
long square(long x){ return x*x }
```

and another file might call square with an int parameter:

```
int i = 4;
i = square(i);
```

The TopSpeed Project System's linker detects your error, and warns you at link time with the message:

```
Warning: _square: type inconsistency, files
         involved are DECLARE and CALL
```

The warning helps you to avoid making an incorrect program that might crash, and is also recorded in the .MAP file.

A Simple Project File

Selecting Project Edit opens editor window 9 with the current project file. A simple project file might look like this:

```
#system auto exe
#model small jpi

#pragma debug(vid=>full)

#compile %main
#link %prjname
```

A project file is made up of a sequence of commands. Each command is introduced by a keyword, and all keywords start with a # character to make them stand out. In the example above, there are 5 commands, and each command starts at the beginning of a line (this is not a requirement Æ project files are free-format).

The first command,

```
#system auto exe
```

indicates the target operating system and file type for this project. This command would normally appear at the start of each project file. The auto indicates that the target operating system is to be the same as the development operating system, while exe indicates that an executable file is being built (other possible target file types are dll or lib).

The second command,

```
#model small jpi
```

indicates the memory model. This command normally follows immediately after the #system command. In this case, we have selected the *small* memory model, using the JPI calling convention of passing parameters in registers.

The third command,

```
#pragma debug(vid=>full)
```

specifies an option to be used in subsequent compilations. #pragma commands can appear anywhere in the project file after the #model command, but it is usual to find most of them at the top of the file, to apply to all compilations.

The fourth command,

```
#compile %main
```

specifies that the file indicated is to be compiled. The % character is used to introduce a macro name to be substituted. The macro main is set by the TopSpeed Environment, to indicate the name of the main source file (how this is determined is described later in this chapter).

The #compile command first determines (by looking up the extension of the nominated source file) the action that is required to compile it. Secondly, by locating the corresponding object file (if any), and examining the version stamps that it contains, the command determines whether recompilation is required.

Files will be recompiled if any of the following are true:

- The object file does not exist,
- Any source file used to build the old object file has been changed,
- The pragma settings used to build the old object file have changed.

These factors are determined automatically by the examination of the object file.

Finally, the command:

```
#link %prjname
```

indicates that the objects specified using #compile, together with any libraries and other objects required, are to be linked to form an executable file. The name of the file to be produced is specified as the value of the macro %prjname. This macro is also set by the TopSpeed Environment, as described below.

The Environment Interface

When using the TopSpeed Environment, project files are configured using the Project menu, and are invoked using the Compile, Link, Make, Run, or Vid commands (see below).

Invoking Projects (Make Mode)

The normal mode of invoking the Project System from the TopSpeed Environment is in *make mode*. In this mode, all commands in the project file are active, and (assuming no errors are encountered) the entire file is processed, to make an up-to-date executable file or files from the nominated sources. The Project System determines the minimum amount of compilation and linking required in order to ensure that the executable file is up-to-date.

Make mode is invoked by selecting the Make menu item (or by using the short-cut Alt-M). Make mode is also entered when a Project is remade automatically through the selection of Run or Vid.

The TopSpeed Environment prompts you for the name of the file to be made. At this point, you can specify either the name of a project file, or the name of the main source file. The environment first assumes that the name is of a project file. If such a file cannot be found, the name is then assumed to be that of the main source file, and the project file UNNAMED.PR is used.

The macros %main and %prjname are set to the assumed names of the main source file and executable filename respectively. In make mode, where a named project file is used, both %main and %prjname are derived from the project filename with path and extension removed. Where UNNAMED.PR is used, both are derived from the name specified for the main source file.

Invoking Projects (Compile Mode)

An alternative mode of invoking the Project System is in *compile mode*, by selecting the Compile menu item (or by using the short-cut Alt-C). In this mode, a single source file is compiled unconditionally, without checking if it is up to date, and the project file is used simply to determine the appropriate options and memory model to use in the compilation.

If an editor window is open when Compile is selected, then the file in that editor window is compiled. If no window is open, you are prompted to supply the name of the file to compile. The project file used is always the current project file (i.e., the project file used for the last compilation, or last selected using the Project menu).

In compile mode, the macro %main is set to the name of the file being compiled. %prjname is set to the assumed name of the executable file, derived from the project file (or from the main source name if UNNAMED.PR is used), in the usual way. The macro %compile_src is also set to the name of the file to be compiled. In compile mode, #link, #dolink, #implib, and #message commands have no effect, nor do #compile commands for files other than the file to be compiled.

Invoking Projects (Link Mode)

The third way of invoking a project file is in *link mode*. This is selected by choosing Link from the main menu (or by using the short-cut Alt-L). Link mode behaves similarly to make mode, except that version checks are not made to ensure that no source files or options have been changed. Therefore, a source file is only re-compiled if the corresponding object does not exist.

Link mode is primarily intended for use in large projects, where the time overhead involved in the safe version-checking employed by the TopSpeed Project System can become noticeable. In such cases, you may prefer to take on the responsibility of keeping object files up to date yourself, and use link mode rather than make mode to create executable files where you are confident that no compilations are required.

Editing Project Files

Simple project files can be edited by means of the Project menu of the TopSpeed Environment. The current project file can be nominated by means of the Project New project menu command - other menu entries on the menu change to reflect the settings in the current file.

When an option is selected from the Project menu, or from one of the associated sub-menus, the current project file is modified to record the selected option. All future compilations which use this project file are affected by this option. Compilations using other project files are unaffected.

Menu editing of project files only affects the portion from the top of the file to the first non-editable command. Only the commands #system, #model and #pragma are editable. Thus, if your project file starts with a command other than one of these, the Project menu will not be able to determine what the current settings of project options are. Also, if pragmas are altered part way through a project file, then the changes will not be reflected in the menu state, and cannot be edited by means of the menu.

The #noedit Command

As a special case, the command #noedit can be placed at the top of a project file. It does not count as a non-editable command, because it does not prevent the menu-interface from examining the project file beyond it. This means that the menu still reflects the correct pragma settings in the project options. However, if #noedit is specified in this way, the menu cannot be used to modify the settings in the project file, and the file text must be edited directly in order to make changes.

The Batch Interface

The Project System can also be invoked directly from the command line using the TSC program. As with the Environment interface, the command

line can be used to invoke the Project System in make, compile or link modes. See Chapter : '*TopSpeed from the Command Line*' for further details.

The Project Language

A project file consists of a sequence of commands, which are processed sequentially by the Project System. Commands are made up from keywords (which all begin with #), strings, and symbols.

The following keywords are recognized by the Project System :

#abort	#expand	#older
#and	#file	#or
#autocompile	#getkey	#pragma
#compile	#if	#prompt
#declare_compiler	#ignore	#run
#dolink	#imlib	#rundll
#edit	#include	#scan
#else	#link	#set
#elsif	#message	#split
#endif	#model	#system
#error	#noedit	#then
#exemod	#not	#to
#exists		

The following characters form symbols in the project language, and may only occur in a project file where the syntax of the project command in question permits, or within quoted strings:

() < > , / =

A string consists of a sequence of characters, delimited by spaces, or by any of the special characters mentioned above. A string may also be enclosed in single or double quotes, which may include any characters. The quotes do not form part of the string.

Note: The Project System is case sensitive Æ upper and lower case are always distinct.

A comment is introduced by two “-” characters, and terminated at the end of the line.

Macro Expansion

A sequence of characters enclosed by % characters indicates a macro name. The second % may be omitted provided that the character following the macro name is not one of the characters below. The following characters are permitted in macro names:

A B C D E F G H I J K L M
 N O P Q R S T U V W X Y Z
 a b c d e f g h i j k l m
 n o p q r s t u v w x y z
 0 1 2 3 4 5 6 7 8 9 _

Whenever a macro name is encountered, it is replaced either by the string which has been associated with that macro, or by an empty string if there is no such associated string.

Two adjacent % characters may be used when a % character is required in a string. This is most often used for delaying macro expansion. For example:

```
#set echo = '#message % %mymac'

#set mymac = 'Hello'
%echo
#set mymac = 'World'
%echo
```

executes the commands:

```
#message Hello
#message World
```

If a single % had been specified in the first #set command, the macro %mymac would have been expanded (to the empty string) before defining the replacement text for the macro %echo.

The #set Command

```
#set string = string
```

The #set command associates the macro name specified by the first string with the second string. Any previous setting for the given macro is lost.

Note: The macro name on the left should not be delimited by % characters.
For example:

```
#set cwindow = jpi
#set linkit = '#link myfile'

...

#if '%cwindow'=jpi #then
  #pragma link(CS_GRAPH.LIB)
#endif

%linkit
```

Compiling and Linking

The primary purpose of most project files is to create one or more executable or library files, while performing the minimum set of recompilations and relinks necessary to ensure that the target files are consistent with the current versions of the corresponding source files.

A typical project file, therefore, consists of a number of commands to specify the options to be used when compiling and linking, followed by one or more #compile commands and finishing with a #link command.

The Link List

The Project System maintains a list of those files which are to be used as input to the linker the next time an executable or library file is created. This list is known as the *link list*. A filename may be added to the link list using the `#pragma link` command.

For example:

```
#pragma link (mylib.lib)
```

However, it is seldom necessary to use `#pragma link` explicitly, as all the predefined TopSpeed compilers add the resulting object file to the link list whenever a source file is compiled. In addition, when the `#link` command is encountered, all required standard library files, and other object files which are imported by those already on the link list are also added to the list. The link list is cleared after each link.

The #compile Command

```
#compile <File-name> [ #to <file-name> ] [ / <pragma> { , <pragma> } ]  
    { , <file-name> [ #to <file-name> ] [ / <pragma> { , <pragma> } ] }
```

The `#compile` command causes each nominated source file to be compiled (if necessary). The name of the object file may be specified using `#to`. If this is omitted, the name is derived from the source filename, with the extension `.obj`.

Any pragmas specified in a `#compile` command apply only to the single source filename that precedes the `/` character.

The macro `%make` is set to on if a compile is necessary, off otherwise. The macros `%src` and `%obj` are set to the names of the source and object filenames.

Each object file is added to the link list, i.e. there is an implicit:

```
#pragma link( %obj )
```

For example:

```
#compile fred.c #to fred.obj  
#compile george.cpp /debug(vid=>full)
```

It is possible to reconfigure the behavior of the Project System when compiling source files of a given extension using the `#declare_compiler` command (see later in this chapter). This may also be used to declare actions

to be performed for different file extensions Æ for example, to support third-party compilers or preprocessors.

The #link Command

```
#link <file-name>
```

The #link command links together (if necessary) all the files in the link list to the nominated executable or library file. The file type is determined by the extension of the nominated target file, or, if there is no extension, by the file type specified in the most recent #system command. If neither is specified, the default is to produce an executable file.

The #link command differs from the similar #dolink command in that (so far as the Project System can determine), any additional object files required are automatically added to the link list before linking. This includes any TopSpeed library files, and also (via an implicit #autocompile command Æ see below) all modules imported via IMPORT clauses in TopSpeed Modula-2 or TopSpeed Pascal, or via #pragma link statements in TopSpeed C or TopSpeed C++ source files. In addition, #link will determine from the target file type and memory model any additional processing that needs to be applied to the output file, for example, when producing an executable file for Windows or in overlay model.

The action of the #link command may be reconfigured by altering the contents of the *Project Link* section of the configuration file TSPRJ.TXT. The effect of #link is to set the macro %link_arg to the specified filename, then execute the sequence of project file commands in that section of the configuration file.

For certain specialized requirements, the use of #link may be inappropriate - for example, if a specialized startup file is required, or when building library files, where explicit control of exactly which files are included may be preferred. In such cases, the #dolink command should be used.

The #dolink Command

```
#dolink <file-name>
```

The #dolink command takes the object files which have previously been added to the link list, and combines them into an executable or library file (depending on the extension of the nominated target file), if required to keep the target file up to date. No additional files are added to the link list, so all required files must have been specified previously, by means of #pragma link, #pragma linkfirst, #compile, and #autocompile. For simple projects, the use of #link is preferable.

The #autocompile Command

```
#autocompile
```

The `#autocompile` command examines the object files which are currently in the link list, to see which objects they need to be linked with. This would include objects specified using a `#pragma link` in a TopSpeed C or C++ source file, or in the case of module based languages such as TopSpeed Modula-2 and Pascal, imported modules.

Each resulting object file, if not already in the link list, is then compiled (if necessary) and added to the link list. If there is more than one possible source for a given object file, an error is reported. This process is repeated until the link list stops changing.

It is not necessary to use `#autocompile` for simple projects where `#link` is used rather than `#dmlink`, as `#link` performs an implicit `#autocompile`.

The #implib Command

```
#implib <file-name> [ <file-name> ]
```

The `#implib` command is used to create (if necessary) a dynamic link import library file. There are two forms of this command, which operate in slightly different ways. If a single filename is specified, this names an import library file, which is created (if not up-to-date) from the object files on the link list. The object files are scanned and each public function or variable is exported. For example:

```
#pragma link( fred.obj, joe.obj )
#implib mylib.lib
```

In the second form of the `#implib` command, an import library filename and a module definition file (.exp) are both specified, and the library file is created (if not up-to-date) from the symbols named in the definition file. This form of the command is equivalent to using the `tsimplib` utility Æ see the “*TopSpeed Advanced Programmer’s Guide*” for further information.

```
#implib <expfile> <libfile>
```

Using `#implib` in the second form requires you to create and maintain the list of exports ‘by hand’, whereas the first form exports all public names automatically. The use of a module definition file is an advantage if you need to maintain compatibility with previous versions of an interface, and it also allows you to export only the functions which need to be exported.

The Microsoft `implib` utility must be used if you require that the .lib file created be acceptable to the Microsoft linker.

Warning: When making DOS DLLs, a feature of the TechKit⁰, a module definition file must be created.

The #exemod Command

```
#exemod <file-name> <file-name> <file-name>
```

This command is the equivalent of using the `tsexemod` utility, required to make advanced overlay model programs, Windows programs and DOS DLLs.

It is not necessary to use this command explicitly when making overlay model programs, unless the advanced features available to the TechKit[®] user are required. This command's facilities are documented fully in the *“TopSpeed Advanced Programmer's Guide”*.

Compiler and Linker Options

Whenever a file is compiled or linked, the current settings of the compiler or linker options are compared to those used when the file was last compiled or linked, to determine whether the file is up to date. If a compilation or link is necessary, the current settings are passed on to the compiler or linker to determine its behavior appropriately.

Compiler and linker options are specified by means of the `#system`, `#model` and `#pragma` commands.

The #system Command

```
#system string [ string ]
```

The `#system` command is used to specify the target operating system and file type. The macros `%system` and `%filetype` are set to the first and second arguments. The `#system` command will affect the behavior of subsequent `#model` and `#link` commands, and so a `#system` command must have been specified before either of these.

The first argument specifies the target operating system, and may be `dos`, `os2`, `win` (for Windows) or `auto` (to indicate that the target is the same operating system as is currently being run).

The second argument indicates the target file type, and may be `exe`, `lib`, or `dll`. If omitted, `exe` is assumed.

The #model Command

```
#model string [ string ]
```

The `#model` command is used to specify the memory model to be used for subsequent compilations and links. This memory model will continue to be used until modified by explicit pragmas, or by another `#model` command.

The first argument specifies the model (`small`, `medium` etc), and must be present. The second indicates the calling convention, which may be `jpi` or `stack`. If omitted, `jpi` is assumed.

Note: The behavior of the `#model` command may be configured using the *Project Model* section of the configuration file `TSPRJ.TXT`. The effect of a `#model` command is to set the macros `%model` and `%jpicall` to its first and second parameters respectively, and then execute the sequence of project commands found in the *Project Model* section of the configuration file.

The `#system` command must be specified before the first `#model` command.

The #pragma Command

```
#pragma <pragma> { , <pragma> }
```

The `#pragma` command modifies the state of the pragma options which affect the behavior of the TopSpeed compilers or linker. The syntax and meaning of all pragmas are discussed in Chapter . The special macro `%pragmastring` expands to the current state of all pragma options which are not in their default state - this can be useful for determining exactly which options are being used for a given compilation. For example:

```
#message '%pragmastring'
```

will display the current pragma state in the message window.

The #ignore Command

```
#ignore <file-name>
#ignore pragmastring
```

There are two forms of the `#ignore` command. The first, where a filename is specified, tells the Project System to ignore the date of the nominated file when deciding whether or not to compile. This is useful when a 'safe' change is made to a widely used header file, to prevent mass recompilation.

The special form `#ignore pragmastring` directs the Project System to ignore the pragma settings when deciding whether or not to compile a file. This may be useful, for example, when a new compile-time macro has been defined, but there is no need to recompile everything.

Flow of Control

Project file commands may be executed conditionally, using `#if`, `#else` and `#elsif` commands. The syntax of the `#if` command is as follows :

```
#if <boolean-expression> #then
  commands
#elsif <boolean-expression> #then
  commands
#else
  commands
#endif
```

The `#elsif` part may be omitted, or may be repeated any number of times. The `#else` part may be omitted.

The expressions are evaluated in order, until one of them yields true, then the following command sequence is executed. If none of the expressions yield true, and the `#else` part is present, then the commands following `#else` are executed. All other commands are ignored.

The syntax and semantics of boolean expressions are described below.

Boolean Expressions

Boolean expressions used in `#if` and `#elsif` commands are made up from the following boolean operators (listed in order of precedence):

The #or Operator

```
boolean_expression ::= <factor> { #or <factor> }
```

A boolean expression containing one or more `#or` operators yields true if the evaluation of any of the factors yields true.

The #and Operator

```
<factor> ::= <term> { #and <term> }
```

A factor containing one or more `#and` operators yields false if the evaluation of any of the terms yields false.

The #not Operator

```
<term> ::= #not <term>
```

A term preceded by the `#not` operator yields true if the evaluation of the term yields false, and vice versa.

The Comparison Operator

```
<term> ::= string = string
```

A term containing a comparison operator yields true if the strings are identical, otherwise false. `==` may be used instead of `=`.

The `=` operator and second string may be omitted, in which case the first string is compared against the string “on”.

The first string may be replaced by an expression of the form `name1(name2)`, where `name2` names a pragma of class `name1`. In this case, the expression is replaced by the current setting of the specified pragma, before the comparison is made.

The #exists Operator

```
<term> ::= #exists <file-name>
```

A term containing the `#exists` operator yields true if the file exists (after applying redirection to the filename), otherwise false.

The #older Operator

```
<term> ::= <file-name> #older <file-name> { , <file-name> }
```

A term containing the `#older` operator yields true if the first file specified is older than at least one of the other files specified, otherwise false. Redirection is applied to all filenames. This operator is often useful to

determine whether a post/pre-processing action needs to be performed. For example:

```
#if mydll.lib #older mydll.exp #then
```

Parenthesized Boolean Expressions

```
<term> ::= ( <boolean-expression> )
```

A term may consist of a *parenthesized boolean expression*, in order to alter or clarify the binding of other boolean operators. The term yields true if the enclosed boolean expression yields true. Arbitrarily complex boolean expressions may be formed.

File System Commands

These commands are all prefixed by the keyword `#file`, and handle the interface to the DOS/OS2 file system with TopSpeed redirection built-in. Redirection applies to all filenames mentioned. The syntax and semantics of the file system commands are as follows:

The #file copy Command

```
#file copy <src-filename> <dst-filename>
```

This command causes a file to be copied from `<src-filename>` to `<dst-filename>`. Both `<src-filename>` and `<dst-filename>` must be filenames without wildcard characters. Redirection is applied to both filenames.

The #file delete Command

```
#file delete <filename>
```

This command causes the nominated file to be deleted. `<filename>` must be a filename without wildcard characters. Redirection is applied to the filename.

The #file move Command

```
#file move <src-filename> <dst-filename>
```

This command moves (renames) a file from `<src-filename>` to `<dst-filename>`. Both filenames must specify files on the same drive. Redirection is applied to both filenames.

The #file touch Command

```
#file touch <filename>
```

This command sets the date and time of `<filename>` to be the current date and time.

The #file append Command

```
#file append <filename> <string>
```

This command appends the specified string to <filename>, followed by a CR/LF pair. The file will be created if it does not exist. This command can be used to build up log-files.

The #file redirect Command

```
#file redirect [ <filename> ]
```

This command changes the current redirection file to <filename>. If no filename is specified, then the redirection file is restored to its value on entry. At the end of the project file, the redirection file is restored to its previous state.

The #file adderrors Command

```
#file adderrors <filename>
```

This command processes the error messages in the nominated file, and adds them to the errors that will be reported when the project terminates.

Each error message must in one of the following formats :

```
(filename lineno,colno): error string  
(filename lineno): error string  
filename(lineno): error string
```

To capture errors from a program with a different error format, a filter program can be used to translate them. For example:

```
#run 'masm %f; > %f.err'  
#file adderrors %f.err  
#run 'myprog %f; | myfilter > %f.err'  
#file adderrors %f.err
```

If any errors are detected, and abort mode is on (see the #abort command below), the project will terminate and the errors will be reported in the error editor window.

The macros %errors and %warnings are set to the number of errors and warnings detected.

The #file prompt Command

```
#file prompt <promptstring> [ <defaultfilename> ]
```

This command prompts for a filename in the same way that the TopSpeed Environment does. If wildcards are entered then file selection windows are opened to choose the required file. The file entered or selected is returned as the value of the macro %reply. For example:

```
#file prompt 'File to run: ' *.exe  
#run %reply
```

Editor System Commands

These commands are all prefixed by the keyword #edit, and handle the interface to TopSpeed Environment Editor. These commands have no effect when a project is executed from the Batch System.

The #edit save Command

```
#edit save <filename> [ #to <filename> ]
```

This command causes the nominated file <filename> to be saved if it is present in any editor window. You are not prompted to confirm that the file should be saved. If #to is specified, the file can be saved with a different name.

The #edit saveall Command

```
#edit saveall
```

This command causes all edited files in all windows to be saved, without prompting.

The #edit savewin Command

```
#edit savewin <winnum>
```

This command causes the contents of the specified window to be saved. <winnum> must specify a number between 0 and 9. This command is generally used to save system editing windows 0 or 9, before loading another file into them (see below).

The #edit loadwin Command

```
#edit loadwin <winnum> <filename>
```

This command loads the nominated file into the specified window, and moves the window to the top of the window stack. If the window previously contained an edited file, the contents (and any edits) are lost.

The #edit refresh Command

```
#edit refresh <filename>
```

This command causes the editor to reload the specified filename from disk if it is present in any editing window. This is usually called when a file has been changed, to ensure that the latest version is on display in the editor.

Note: Changes to the file within the editing window will be lost, so the command should be used with care.

The #edit open Command

```
#edit open <winnum> [ <filename> ]
```

This command terminates the current project and enters the specified editor window. If a filename is specified, then it will be loaded into the editor window. If window 0 is specified and no filename is supplied, the error editor will be entered in the first file to have errors reported on it.

Miscellaneous Commands

The #message Command

```
#message <string>
```

This command displays the specified string in the make display window. This can be used to indicate progress through the project file, or to display status messages.

Messages are only displayed in make or link mode. When running under the Batch System, messages are suppressed if the *quiet* command-line switch is specified (/zq). For example:

```
#message "finished making %prjname"
```

The #error Command

```
#error <string>
```

This command terminates the current project. Under the TopSpeed Environment, the error editor is opened at the position of the #error command, and displays the supplied string as the error message. Under the Batch System, the error is displayed in the standard TopSpeed format. For example:

```
#if "%name"="" #then  
  #error "name not set up"  
#endif
```

The #abort Command

```
#abort [ on | off ]
```

This command is used to control whether a failed compilation or #run command will terminate a project. If abort mode is on, a project will be aborted as soon as a compilation fails, or a #run command produces a non-zero return-code. If abort mode is off, a project will only be aborted if an internal command fails, including a #link, #implib or #exemod command.

#abort on will set abort mode to on, while #abort off will turn it off. #abort without one of the above arguments will abort the current project immediately.

The default abort mode is on when running under the TopSpeed Environment, and off when run from the command line.

The #prompt Command

```
#prompt <promptstring> [ <defaultstring> ]
```

This command prompts you to enter a string, displaying <promptstring>. If <defaultstring> is specified, this will be used as the default value presented. The string you enter is returned as the value of the macro %reply. For example:

```
#prompt "Command line: " %cline
#set cline = %reply
```

The #getkey Command

```
#getkey <promptstring> [ <keysstring> ]
```

This command displays the specified prompt string, and waits for one of the keys specified in <keysstring> to be pressed. If <keysstring> is not specified, this command waits for Escape to be pressed.

The macro %reply is set to the character of the key pressed. For example:

```
#getkey 'Abort/Retry/Quit?' 'ARQ'
#getkey "Errors found, press escape"
```

The #run Command

```
#run <commandstring> { <runoption> }
```

This command executes the command specified by <commandstring>. Various options may be specified after the command string, from the following list:

pause	prompts for escape to be pressed after the program finishes executing, before continuing executing the project file.
timed	times the execution of the command, and displays the elapsed time before continuing.
Note:	Built-in commands (for example, DIR, COPY etc.) cannot be used with this option.
swap	swaps the TopSpeed Environment out of main memory before executing the command (DOS only).
fail_abort	aborts the project if the program returns a non zero DOS return value.
rte_abort	checks for the occurrence of run time errors, and will abort the project if they occur, and try to find the error in the source (if debug information is turned on).
Note:	Built-in commands (for example, DIR, COPY etc.) cannot be used with this option.
no_abort	non-zero return codes will not cause the project to be aborted.
session	under OS/2 this will cause the program to be run in a separate OS/2 session. This command has no effect when used under DOS.
no_window	will not clear the screen before executing the file. Any output from the program will garble the screen.

For example:

```
#run "dir > dir.log"
#run "myprog" pause rte_abort
```

Note: Filenames within the command string (with the exception of the executable filename itself) are not automatically subject to redirection Æ #expand may be used before using #run if this is required.

The use of the no_abort, rte_abort and fail_abort options overrides the current setting of the #abort switch (see above).

The #include Command

```
#include <file-name>
```

A copy of the contents of the nominated file is inserted in the input stream. <file-name> should specify a valid filename, which will be redirected using the current redirection file.

The #expand Command

```
#expand <file-name>
```

The filename is subjected to redirection analysis, and the following macros are defined:

%cpath	is set to the fully expanded filename where the file would be created.
%opath	is set to the fully expanded filename where the file would be opened.
%ext	is set to the extension of the filename.
%tail	is set to the filename, less extension, drive and path.
%cdir	is set to the directory where the file would be created.
%odir	is set to the directory where the file would be opened for read (if the file does not exist %opath is set the same as %cpath).

For example, suppose the redirection file has the line,

```
*.def : . ; c:\ts\include
```

and the file c:\ts\include\io.def exists, and the current directory is d:\test then,

```
#expand io.def
```

is equivalent to,

```
#set opath = d:\test\io.def
#set cpath = c:\ts\include\io.def
#set ext   = .def
#set tail  = io
#set odir  = d:\test\
#set cdir  = c:\ts\include\
```

The #split Command

```
#split <file-name>
```

The filename is split into its base and extension. The following macros are defined:

```
%ext is set to the extension of the filename.
%name is set to the filename, less extension.
```

For example:

```
#split d:\name.exe
```

is equivalent to,

```
#set ext    = exe
#set name   = d:\name
```

The #scan Command

```
#scan <file-name>
```

This command may be used to speed up a project where a large number of source files are involved. All files found via redirection which match the nominated filename (which should include wildcards), are added to the Project System's file date cache. If only a small percentage of the files visible (via redirection) are actually used in the make, using #scan may not be worthwhile. For example:

```
#message "Scanning..."
#scan *.def
#scan *.mod
#message "Done"
```

The #declare_compiler Command

```
#declare_compiler string = string
```

This defines a macro (the second string) which is invoked when compiling source files with an extension matching the first string. The macros %src and %obj are set to the names of the source and object files.

Generally, you will not have to use this command explicitly, as all TopSpeed compilers (and other common cases such as MASM) are pre-declared in the *Project Predefined* section of the configuration file TSPRJ.TXT, which is implicitly included at the top of every project. For example, the following is an extract from the standard TSPRJ.TXT:

```
#declare_compiler mod=
' #set make = %%remake_jpi
  #if %%make #then
    #rundll TSMOD %%src %%obj
  #endif
  #pragma link (%%obj)
  #set tsm2=on
,
```

The #rundll Command

```
#rundll string string string
```

This command invokes an integrated JPI compiler/utility. The first string is the DLL name, the second is the source filename, and the third is the output filename.

You should never have to use this command explicitly, as all JPI compilers/utilities are pre-declared in the *Project Predefined* section of the configuration file TSPRJ.TXT, which is implicitly included at the top of every project.

Error and Warning Macros

The commands #rundll and #file adderrors set the values of the macros %errors and %warnings to indicate the number of errors and warnings detected.

These macros may be inspected in order to determine the status of the make after one of the above commands:

```
#if #not %warnings=0 #then
#getkey "Press E to edit, C to continue" "EC"
#if %reply = E #then
#edit open 0
#endif
#endif
```

Reconfiguring the Project System

The behavior of the TopSpeed Project System can be modified in a number of ways by changing the configuration file TSPRJ.TXT. The TopSpeed System is supplied with a standard version of this file, but you may wish to configure the behavior to suit your own particular requirements or working habits. If you alter TSPRJ.TXT, you must run TSCFG for the changes to take effect.

The file TSPRJ.TXT is divided into 3 sections, headed by a directive *Project Predefine*, *Project Model* or *Project Link*, and terminated by the next such directive or by *Project End*.

Note: Lines in the configuration file that start with an exclamation mark are directives to the configuration program TSCFG. Lines beginning with a double exclamation mark are comments that the configuration program will strip.

Predefined Compilers

The *Project Predefine* section of the configuration file extends from the *Project Predefine* directive until the following Project directive. The contents of this section (after stripping lines introduced by a double exclamation mark) are included implicitly at the head of every project.

This section is normally used to declare the actions required when compiling source files with different extensions. If you have other compilers or preprocessors that you wish to use with the TopSpeed system, you can add a `#declare_compiler` command for the relevant extension here, rather than adding it to every project file explicitly. The supplied `declare_compiler` commands may be used as examples.

The special macros `%remake` and `%remake_jpi` may be used in a `declare_compiler` command to determine whether a recompilation is necessary. These macros examine the files nominated by the macros `%src` and `%obj`, to determine whether a recompilation is necessary. If it is, a message is printed in the make display window. The `%remake` macro determines whether a remake is necessary based simply on the two file dates, while `%remake_jpi` will examine the object file (which should have been produced by a TopSpeed compiler or assembler) in order to determine dependencies on other source files and on pragma settings. Both macros will return on to indicate that a remake is required, off otherwise.

Memory Models

The *Project Model* section of the configuration file extends from the *Project Model* directive until the following *Project* directive. The contents of this section are included implicitly whenever a `#model` command is executed in a project.

If you examine the standard `TSPRJ.TXT` file, you will see that the TopSpeed memory models are in fact simply collections of pragma settings. The compilers themselves do not have the concept of 'small model' or 'medium model'. If you wish to define your own specialized memory model, this is where it can be done.

The standard *Project Model* section makes use of various Project System macros - `%system` and `%filetype` are set by the `#system` command, and `%model` and `%jpicall` are set by the `#model` command before including the *Project Model* section. Other macros are set by the *Project Model* section to be used by the *Project Link* section. A list of all macros that are used for special purposes by the Project System is given later in this chapter.

Linking

The final configurable section is headed *Project Link*. This section is used to control the behavior when a `#link` command is executed. It automatically selects the relevant startup files, additional object files, library files and additional processing required to successfully perform the link. The macro `%link_arg` specifies the filename given as argument to the `#link` command.

If you have defined your own memory model in the *Project Model* section, as described above, you will also probably want to add the relevant commands to the *Project Link* section to include a suitable library.

Special Project Macros

A number of macros are used for special purposes by the Project System, and you should avoid defining macros of the same name inadvertently. Similarly, you should not define macros using trailing underbars.

The following is a list, in alphabetical order, of all such macros:

<code>%action</code>	set to make, link, compile or run, depending on the mode of invocation.
<code>%C</code>	set by the <code>#model</code> command to either F to indicate the stackframe calling convention or an underbar to indicate the standard TopSpeed calling convention. This macro is used by the <code>#link</code> command when determining library filenames.
<code>%cdir</code>	set by the <code>#expand</code> command.
<code>%cgraph</code>	set to either <code>jpi</code> or <code>borland</code> , to indicate which version of the C graphics library to use. This macro is set automatically when you include one of the relevant header files, and is used by the <code>#link</code> command to determine what library to include.
<code>%compile_src</code>	in compile mode, this is set to the name of the file to be compiled, with path and extension where available. Otherwise, it is set to the empty string.
<code>%cpath</code>	set by the <code>#expand</code> command.
<code>%cwindow</code>	set to either <code>jpi</code> or <code>borland</code> , to indicate which version of the C text windowing library to use. This macro is set automatically when you include one of the relevant header files, and is used by the <code>#link</code> command to determine what library to include.
<code>%devsys</code>	set by the TopSpeed Environment to <code>dos</code> , <code>os2</code> or <code>win</code> ,

	depending on the development operating system, and examined by the <code>#system</code> command.
<code>%editfile</code>	set to the name of the file being edited in the topmost window. If no window is open, or in batch mode, it is set to the empty string.
<code>%editwin</code>	set to the window number (0-9) of the topmost window. If no window is open, or in batch mode, it is set to the empty string.
<code>%errors</code>	count of errors produced by preceding compilation or <code>#file adderrors</code> command.
<code>%ext</code>	set by the <code>#split</code> and <code>#expand</code> commands.
<code>%filetype</code>	set by the <code>#system</code> command to its second argument, and examined by the <code>#link</code> command.
<code>%jpicall</code>	set by the <code>#model</code> command to its second argument, and examined by the <code>#link</code> command.
<code>%link_arg</code>	set to its argument by the <code>#link</code> command.
<code>%M</code>	set by the <code>#model</code> command to a single letter to indicate the memory model in the filename of TopSpeed library files, and examined by the <code>#link</code> command.
<code>%main</code>	set to the assumed name of the main source file. In make or link mode when not using UNNAMED.PR, this is derived from the project filename, with path and extension removed. Otherwise, it is the supplied source filename complete with path and extension if specified.
<code>%make</code>	set to on or off by the <code>#compile</code> , <code>#link</code> and <code>#dolink</code> commands, to indicate whether the target file was up to date.
<code>%model</code>	set by the <code>#model</code> command to its first argument, and examined by the <code>#link</code> command.
<code>%name</code>	set by the <code>#split</code> command.
<code>%O</code>	set by the <code>#model</code> command to W, P or R to indicate Windows, OS/2 (<i>Protected mode</i>) or DOS (<i>Real mode</i>). The <code>#link</code> command uses this to derive the name of any required library files.
<code>%obj</code>	set to the object filename in a <code>#compile</code> command.
<code>%odir</code>	set by the <code>#expand</code> command.
<code>%opath</code>	set by the <code>#expand</code> command.
<code>%pragmastring</code>	will always expand to the current state of the pragma settings Æ this is useful for debugging.
<code>%prjname</code>	set to the assumed name of the project - this is usually

derived from the project filename, but with the path and extension removed. Where UNNAMED.PR is being used, it is derived from main source filename without source and extension.

%remake	used within declare_compiler macros to determine whether source/object dependencies require a remake.
%remake_jpi	used within declare_compiler macros to determine whether source/object dependencies require a remake. %remake_jpi should be used for object files created by TopSpeed compilers, which contain additional information.
%reply	set by the #prompt and #getkey commands.
%src	set to the source filename in a #compile command.
%system	set by the #system command to its first argument, and examined by the #model and #link commands.
%tail	set by the #expand command.
%tsc	set to on if a C or C++ source is compiled.
%tscpp	set to on if a C++ source is compiled.
%tsm2	set to on if a Modula-2 source is compiled.
%tspas	set to on if a Pascal source is compiled.

The above macros are examined by the #link command to determine which libraries to include, and then set to off.

%tsmode	set to either bat or env, depending on whether the Project System was being invoked in batch mode or from the TopSpeed Environment.
%warnings	count of warnings produced by preceding compilation or #file adderrors command.

CHAPTER 3

TopSpeed from the Command Line

The TopSpeed system may be invoked in two ways. In most cases, the TopSpeed Environment TS.EXE will be the normal mode of invocation, but in some cases, it is more convenient to use the command line method, using the file TSC.EXE.

The TopSpeed *Batch System* works in a slightly different manner to the command-line interfaces supplied with many compilers, so it is worth taking a little time to read the overview below if you are unfamiliar with the TopSpeed System.

Command Line Overview

The TSC command should be invoked with a *command tail* containing a list of filenames and command-line options, separated by spaces, commas or tabs. The command line is not sensitive to the order of the options and filenames specified Æ all options specified will apply to all filenames.

Command-line options are introduced by a slash /, and are terminated by the next space, comma or tab character. Any item on the list of parameters which is not introduced by a slash character is taken to be a filename.

Just as the Project System can be invoked in three modes (*compile*, *link* or *make*), the TopSpeed Batch System can also be invoked in the same three modes.

Compile Mode

If neither the command-line option /M nor the command-line option /L appear on the command line, the Batch System is invoked in compile mode. In this mode, all filenames specified are taken to name source files, and each in turn is compiled, using any options specified on the command line.

If a project file is nominated using the /FP command-line option (see below), then this project is executed, in compile mode, for each file in turn. Options specified on the command line may be overridden by those specified within the project file. If no project file is specified, then only the options specified on the command line are used.

Note: In the Batch System, unlike when working from the TopSpeed Environment, no 'default' project file is used.

Make Mode

If the command-line option */M* appears anywhere on the command line, and the option */L* does not, the Batch System is invoked in *make mode*. In this mode, each filename specified is processed in one of two ways:

- If the filename names a project file, or if a project file with the same name (ignoring path and extension) as the nominated file is found via redirection, the project file is executed in make mode. Options specified in the project file may override those specified on the command line.
- If no project file is found, the filename is taken to name a source file. The corresponding object file is linked, together with any other required object files or libraries, to form an executable file. Any object file that does not exist, or which is not consistent with the date stamps of its constituent sources or with the specified compiler options, is recompiled using those options. The executable filename is derived from the source filename with the extension *.EXE*, and with the pathname derived from the redirection file.

In the second case, the options specified on the command line are used whenever a file is recompiled. Note that, unlike when working from the TopSeed Environment, no ‘default’ project file is used.

Link Mode

If the command-line option */L* appears anywhere on the command line, the Batch System is invoked in *link mode*. Filenames specified are treated in the same way as for make mode (see above), except that the project file (if any) is executed in link mode rather than in make mode. The distinction between the two modes is described in detail in Chapter : ‘*The Project System*’. The main difference is that link mode ignores file date dependencies when determining whether to recompile a file, and only does so if the corresponding object file does not exist.

If no project file is found, the filename is taken to name a source file. The corresponding object file will be linked, together with any other required object files, to create an executable file. Any object files that do not exist will be created by recompiling the corresponding source. However, unlike in make mode, existing object files are not checked for consistency with their constituent source files or the specified compiler options.

Command-line Options

This section summarizes TSC's command-line options. There are two kinds of option:

- Options that are interpreted by the command line processor TSC itself. These options control how TSC searches for files or displays messages.
- Options that are interpreted by the Project System, and/or passed on by it to the compilers and linker. These options correspond to pragmas and memory model commands that can be specified in project files.

Note: You **must not** have any spaces between an option and its argument, or within the argument. Also, options themselves are case insensitive, but the arguments to options such as /d and /s are case sensitive.

TSC Options

The general compiler options apply to the Project System and to the compilers, and specify how files are located, and what output is generated.

/i<dirname> selects the directory for include files, instead of using the TopSpeed redirection file. For example:

```
TSC JESPER.C /iC:\TSC\PM\INCLUDE
```

will compile JESPER.C searching the directory C:\TSC\PM\INCLUDE for include files. This overrides the use of the redirection file.

/y specifies that the DOS environment variables INCLUDE, LIB, OBJ, TMP and PRJ should be used to indicate where to locate files (instead of using the TopSpeed redirection file). For example, an environment such as:

```
INCLUDE = C:\TSC\PM\INCLUDE
OBJ = C:\TSC\OBJ
LIB = C:\TSC\PM\LIBRARY
TMP = E:\
PRJ = E:\TSC\PR
and a command line of:
TSC JESPER.PR /y /m
```

will make the project C:\TSC\PRJ\JESPER.PR using C:\TSC\PM\INCLUDE for the include files and C:\TSC\PM\LIBRARY for the libraries, while storing any temporary files on drive E:.

/fp<name> indicates that the nominated project file should be used. Any options specified in this project file will override

	options specified on the command line.
/fr<name>	indicates that the nominated redirection file should be used. If this command is not specified, and the /y and /i options are not specified, then the redirection file TS.RED will be used, if found. TSC will search for this redirection file using the PATH environment variable, and also in the directory from which the TSC command was started.
/zq	sets quiet mode. This suppresses the display of all messages except for error messages. This allows you to use TSC with editors, such as <i>Brief</i> , which require a file containing only error messages.

Compilation and Project Options

The compilation command-line options only affect the behavior of the compilers and linker. A few, such as ANSI type-checking for C and range checking for Modula-2, are language-specific. Most apply to all languages. They control such features as compile- and run-time checking, code generation and memory model selection.

Each option is documented in full under the relevant pragma or Project System option.

/a[+ -]	ANSI C keywords only. See pragma option(ansi).
/b[+ -]	generate line number information in object file. See pragma debug(line_num).
/cc[+ -]	constants in code. See data(const_in_code) pragma.
/d<id><=<text>	define macro for preprocessing. See pragma define.
/e[+ -]	enable/disable language extensions. See pragma option(lang_ext).
/f[+ -]	select stack frame calling convention. See Project System command #model.
/j[+ -]	set default char type to unsigned. See pragma option(uns_char).
/m<model>	select memory model. <model> may be one of the following: <ul style="list-style-type: none"> s small model c compact model m medium model l large model x extra-large model t multi-thread model o overlay model

d dynalink model

See the Project System command #model.

/n[+|-]

allow nested comments. See pragma option(nest_cmt).

/o<option>

set optimization option. <option> may be one of the following:

- x all optimizations
- t time optimizations
- c constant optimizations
- j jump optimizations
- a alias optimizations
- f stack frame optimizations
- e common sub-expressions
- h peephole optimizations
- l loop optimizations
- r register optimizations

See pragma optimize.

/op<id>

select target processor. <id> may be one of the following:

- 0 8086
- 2 80286
- 3 80386
- 4 80486

See pragma optimize(cpu).

/oq<id>

select target coprocessor. <id> may be one of the following:

- 0 emulator
- 1 8087
- 2 80287
- 3 80387

See pragma optimize(copro).

/p[+|-]

generate preprocessor output only. See pragma option(pre_proc).

/pc[+|-]

retain comments in preprocessor output. See pragma option(incl_cmt).

/pl[+|-]

minimize number of blank lines in preprocessor output. See pragma option(min_line).

/r

remake all objects unconditionally. This option is only effective in make mode. Normally, object files are only remade if a component source file or compiler option has changed.

/rg[+ -]	guard checking. See pragma check(guard).
/ri[+ -]	array index checking. See pragma check(index).
/rn[+ -]	nil pointer checking. See pragma check(nil_ptr).
/ro[+ -]	overflow checking. See pragma check(overflow).
/rr[+ -]	range checking. See pragma check(range).
/rs[+ -]	stack overflow checking. See pragma check(stack).
/rv[+ -]	variant record checking. See pragma check(field).
/s<name>=<text>	sets the Project System macro <name> to the string <text>. If just <name> is specified, then the replacement string defaults to on. See the Project System command #set.
/t<system>	set target operating system. See the Project System command #system.
/u<name>	undefine predefined compiler macro <name>. See pragma define(<name>=>off).
/v0	generate no VID debug information. See pragma debug(vid=>off).
/v1	generate minimum VID debug information. See pragma debug(vid=>min).
/v2	generate full VID debug information. See pragma debug(vid=>full).
/w[+ -]	enable/disable all warning messages. See pragma warn.
/w*	treat all warnings as errors. See pragma warn.
/w<xxx>[+ -]	control individual warnings. See pragma warn.
/w<xxx>*	treat the warning xxx as an error. See pragma warn.
/zg	generate prototypes. See option(prototypes) pragma.

Using Pragmas on the Command Line

In addition to using the above command-line options, pragmas may be specified on the command line. These take the same form as in a project file, and the same pragmas may be specified, with the following exceptions:

- Since DOS and OS/2 interpret the > character as indicating file redirection, pragmas specified on the command line are specified using the format class(name=value).
- Pragmas requiring a register-list may not be specified on the command line.

For example, the commands

```
TSC richard /v2 /m  
TSC david /w*
```

could equivalently be written

```
TSC richard /debug(vid=full) /m  
TSC david /warn(wall=err)
```

CHAPTER 4

Pragmas

All TopSpeed languages, and the TopSpeed Project System, use a common set of compiler options, and a common syntax for expressing them. These compiler options are known as *pragmas*. In general, all TopSpeed's pragmas may appear in the source code in any TopSpeed language, and may appear in a project file, and the effect will be the same.

Conventions and Syntax

All the pragmas use the following syntax:

Pragma ::=

```
Class '(' Name '=>' Setting { ',' Name '=>' Setting } ')' |
'save' |
'restore' |
'link' '(' File { ',' File } ')' |
'linkfirst' '(' File ')'
```

Class ::=

```
'call' | 'check' | 'data' | 'debug' |
'define' | 'expr' | 'link_option' |
'module' | 'name' | 'optimize' |
'option' | 'warn'
```

Setting ::=

```
BoolSetting | Number | RegList |
String | Ident
```

BoolSetting ::= 'on' | 'off'

RegList ::= '(' Reg { ',' Reg } ')'

Reg ::=

```
'ax' | 'bx' | 'cx' | 'dx' | 'si' | 'di' | 'ds' | 'es' |
'st0' | 'st1' | 'st2' | 'st3' | 'st4' | 'st5' | 'st6'
```

Number ::= decimal or hex number

Ident ::= modula, pascal, c, or c++ identifier

String ::= character string

File ::= filename

Name refers to the actual pragma name which you will find documented below. While all TopSpeed compilers will accept any legal pragma, some will be ignored in certain languages or contexts. For example, the following are valid TopSpeed pragmas:

```
debug(vid=>full)
call(reg_param=>(ax,dx,cx), reg_saved=>())
```

Modula-2 Pragma Syntax

Pragmas in TopSpeed Modula-2 occur in a special form of comment which begins with '(*#'. For example:

```
(*# check( index => off ) *)
```

Old-type Compiler Directives

In the original version of TopSpeed Modula-2, compiler directives starting with a \$ were used to specify compiler options. These directives will still be accepted in this version of TopSpeed Modula-2, with the following exceptions:

- \$B (Ctrl-Break handler). This is no longer supported. Use Lib.EnableBreakCheck instead.
- \$D (data segment name). This is supported, but adds the suffix `_BSS` (for uninitialized data) or `_DATA` (for initialized data) to the name instead of the `D_` prefix.
- \$J (use IRET instead of RET). This is not supported. Instead, you should use the pragma:

```
(*# call( interrupt => on ) *)
```

However, you may find that you have to make other changes as well as the effect of the pragma is different from the \$J directive:

- \$K (C calling convention). This is not supported. Instead, you should use the pragma:

```
(*# call( c_conv => on ) *)
```

- \$M (code segment name). This is supported but adds the suffix `_TEXT` to the name instead of the `C_` prefix.
- \$P (external names for local procedures). This is no longer supported. It is no longer applicable.
- \$Q (procedure tracing). This is no longer supported. Instead, you should use the pragma:

```
(*# debug( proc_trace => on ) *)
```

This enables a different method of tracing procedures. Refer to the `proc_trace` pragma for further details.

- `$X` (80x87 stack spilling). This is no longer supported (and is no longer necessary).
- `$Z` (NIL pointer checks). This still does NIL pointer checks but no longer clears memory.
- `$@` (preserve DS). This is no longer supported.

The support for these directives has been included with later systems so that your old programs and modules will recompile with minimum changes. However, you should avoid using the old directives with new programs, and use pragmas instead.

Pascal Pragma Syntax

Pragmas in TopSpeed Pascal occur in a special form of comment which begins with `'(*#'` or `'{#'`. For example:

```
(*# check( index => off ) *)
{#  check( index => off ) }
```

C and C++ Pragma Syntax

Pragmas are an integral part of the C and C++ languages, and are implemented as compiler directives:

```
#pragma check( index => off )
```

Project System Pragma Syntax

Pragmas in the TopSpeed Project System use a similar syntax to the C and C++ languages:

```
#pragma check(index => off)
```

Pragmas in the Project System may also be specified in the `#compile` command, to apply to a single compilation. For example:

```
#compile mandel.mod /debug(vid=>on)
```

Pragmas

The remainder of this chapter describes all pragmas recognized by the TopSpeed System. The descriptions are organized by the pragma *class*. Each pragma is characterized by its class, name, and the range of values which are acceptable. The pragma descriptions indicate the acceptable values for each pragma, as in the following example:

```
vid => off | min | full
```

This indicates that the vid pragma will accept the values off, min or full. This pragma has class debug, therefore its description is found in the debug pragmas section below.

Unless otherwise specified, all pragmas may be used with the same effect in all TopSpeed languages and in project files.

Call Pragmas

Pragmas with the class name call affect all aspects of calling conventions, code segments, and code pointers. The current settings of the call pragmas at the point at which a function or procedure's *definition* is encountered determines the calling convention that is used to call the function. TopSpeed compilers will detect if an inconsistent calling convention is used when a function is called, and the type-safe linker will report an error if the calling conventions attributed to a given function do not match in every object file.

```
near_call => on | off
```

Specifies whether function calls are near or far. When on, the compiler calls functions with near calls even if compiled in a large code model. The compiler can only use near calls if the calling and called functions are in the same segment. The compiler checks that this is the case.

The default value depends on the memory model; in small and compact the default is on, in all other models it is off. For example:

```
call(near_call=>off)
```

forces the compiler to use far calls independent of the memory model.

```
same_ds => on | off
```

Specifies whether to load the data segment (DS) register on entry to a function. When on, DS will not be loaded as part of the function prolog. This will only be correct when the DS setting of the calling function matches that of the called function. The compiler checks that this is the case.

This option is off by default in the extra large and multi-thread models, and on in the other models. For example:

```
call(same_ds => off)
```

This forces DS to be loaded in the function prolog.

```
c_conv => on | off
```

When on, this option enables the Microsoft C calling convention. In this convention, the compiler pushes function parameters in right to left order on the stack and the caller pops these parameters off the stack. This is not the default, so you should only use this pragma when interfacing to Microsoft C code. For example:

```
call(c_conv=>on)
```

You can also use the `cdecl` keyword in C and C++ for the same effect.

Note: Functions taking a variable number of parameters always use this convention regardless of the setting of this pragma.

See also `standard_conv`, which has the same effect for C and C++, but is ignored for Pascal and Modula-2. The `standard_conv` pragma is set off for jpi calling convention models, and on for stack calling convention models.

```
interrupt => on | off
```

This pragma must be specified before the definition of a function that is an interrupt handler. A special entry sequence is generated which preserves all registers. When using `call(interrupt=>on)` you should also use `call(reg_param=>(), same_ds=>off)`.

The saved machine registers may be declared as parameters. The order of these parameters depends on whether `c_conv` is on or off:

If `c_conv` is on, the order is:

```
ES,DS,DI,SI,BP,SP,BX,DX,CX,AX,IP,CS,Flags
```

If `c_conv` is off the order is

```
Flags,CS,IP,AX,CX,DX,BX,SP,BP,SI,DI,DS,ES
```

The default value for this pragma is off. This pragma is not allowed in project files.

```
inline => on | off
```

If this pragma is set on before a function definition, the compiler makes a copy of the function in the code rather than using a call instruction. The default value is off.

You can use this convention for any function, but you typically use it together with `reg_param` for simple machine-code functions. For example:

```
#pragma save
#pragma call(inline => on, \
            reg_param => (dx,ax))
static void outp(int port,
                unsigned char byt) =
{
    0xEE, /* out dx,al */
};
#pragma restore
```

makes `outp` an inline function, so a call to it appears as a single 80x86 machine instruction: `out dx,al`.

```
seg_name => identifier
```

Specifies the code segment name. `call(seg_name => nnn)` means that the compiler places the code for the function in segment `nnn_TEXT`. The default value depends on the memory model; in small and compact models, the

default is null; in the other models, it is the name of the source file. For example, a code segment named `_TEXT` would be specified as:

```
#pragma call(seg_name => null)
```

and a code segment named `MYCODE_TEXT`, would be specified as:

```
#pragma call(seg_name => MYCODE)
```

The default setting is language dependant, and will not be defined by the Project System.

```
ds_entry => identifier
```

This pragma indicates a segment name to which the DS register will point throughout the execution of a function. If the identifier is null, then the compiler names the segment `_DATA`. If the identifier is none, the compiler does not assume a fixed DS during function execution and uses DS as a general purpose segment register like ES.

```
call(ds_entry => MYDATA)
```

means that on entry to the function, DS will point to the segment named `MYDATA_DGROUP`.

```
reg_param => RegList
```

TopSpeed languages pass function parameters in machine registers rather than using the stack. This generates smaller and faster code. This pragma allows you to fine-tune individual function calls for maximum speed. Other vendors' languages use a less efficient calling convention; you must, therefore, disable this calling convention when interfacing to precompiled objects written for these compilers (see the "*TopSpeed Advanced Programmer's Guide*"). This pragma has no effect on structure parameters, which are always passed on the stack.

The argument for `reg_param` is a register list, specifying which registers should be used. Registers for parameters are allocated left to right from the list. The table shows how the compiler allocates registers dependent on parameter types:

Parameter size	Possible registers
1 byte	ax, bx, cx, dx
2 bytes	ax, bx, cx, dx, si, di
4 bytes	ax, bx, cx, dx, si, di for low word. ax, bx, cx, dx, si, di, es, ds for high word.

Note that the es and ds registers will only be used for the high word of a 4-byte parameter where that parameter is of pointer type.

If either the low or high word cannot be allocated, then the whole parameter is passed on the stack.

floating point	st0, st1, st2, st3, st4, st5, st6
----------------	-----------------------------------

When the compiler exhausts the list of registers, it passes the parameter on the stack. If you specify an empty list, the compiler uses the stack for all parameters.

The default setting for the jpi calling convention is:

```
call(reg_param=>(ax,bx,cx,dx,
                  st0,st6,st5,st4,st3))
```

The default setting for the stack calling convention is:

```
call(reg_param => ())
reg_saved => RegList
```

This pragma specifies which registers a function preserves. The argument `RegList` is a list that specifies the set of registers.

The default set for the jpi calling convention is:

```
call(reg_saved=>(ax,bx,cx,dx,si,di,ds,st1,st2))
```

The default set for the stack calling convention is:

```
reg_saved=>(si,di,ds,st1,st2)
o_a_size => on | off
```

This pragma is available under Modula-2 only. When on, this option passes the size of open array parameters on the stack:

```
(*# call( o_a_size => on ) *)
```

This pragma has no effect for *value* parameter open arrays, unless the `o_a_copy` pragma is set off.

The default setting is on.

```
o_a_copy => on | off
```

This pragma is available under Modula-2 only. When on, open array parameters are copied onto the stack as part of the procedure prolog. If off, only a reference to the array is passed. The default setting is on.

```
iopl => on | off
```

This pragma may not be used in project files and is only available under OS/2. It allows procedures to use input and output instructions by giving the code segment *IO privilege*. The default setting is off.

```
ds_eq_ss => on | off
```

This pragma is available under Modula-2 and Pascal only. It controls whether VAR parameters use 16- or 32-bit pointers. The default setting is on for small and medium models, otherwise off.

```
var_arg => on | off
```

This pragma is available under Modula-2 and Pascal only. When on, it implies that the following procedures take a variable number of arguments. This effectively disables the “too many arguments” error that the compiler

would normally detect. The consequence, however, is that the compiler cannot carry out any type checking on the arguments.

This pragma should be used when calling C procedures (such as printf) where the number of arguments varies:

```
(*# call(var_arg => on,
        reg_param=>()),
   c_conv=>on ) *)
```

The default setting is off.

```
reg_return => RegList
```

This pragma is used to specify the registers to be used for return values of integer, pointer and floating point types. For example:

```
call(reg_return => (bx,cx))
```

The default setting is:

```
call(reg_return=>(ax,dx,st0))
windows => on | off
```

This pragma is used to generate the special entry sequence which Microsoft Windows requires for call-back routines. It only has an effect when `c_conv=>off` and `near_call=>off` are set.

```
result_optional => on | off
```

This pragma is available under Modula-2 and Pascal only. It can be used to call a procedure as a proper procedure without generating a compiler error. For example:

```
(*# save *)
(*# module( result_optional => on ) *)
PROCEDURE FuncProc(x: CHAR): CARDINAL;
(*# restore *)
```

With this declaration, you can write both of the following:

```
i := FuncProc('a');
FuncProc('a');
```

This is only useful when the called procedure has a side effect that is more important than the result. It is particularly useful when calling TopSpeed C library procedures.

The default setting is off.

```
set_jump => on | off
```

This pragma should only be used for the library routines which implement non-local jumps. The effect is to inform the compiler of the non-standard register saving properties of these routines. This pragma may not be used in a project file.

```
overlay => on | off
```

This tells the compiler that the TopSpeed segment-based overlay system (overlay or DOS dynalink model) is being used. The effect is that extra stack

frames may be generated for far routines, and far routine constants are always laid out as double word pointers in memory. The default is on for overlay model, and for dynalink model for DOS executables, and off elsewhere.

```
s_copy => on | off
```

This pragma is available under Pascal only. It controls whether a value parameter of string type is copied inside a procedure. The default setting is on.

This pragma can be used to make a procedure more efficient if the procedure does not modify the string parameter.

```
t_l_size => on | off
```

This pragma is available under Pascal only. It controls whether typeless parameters have their size passed to a procedure. If the size is passed, it is possible to use the size function on the typeless formal parameter to establish the size of the actual parameter. The default setting is off.

```
t_l_copy => on | off
```

This pragma is available under Pascal only. It controls whether value typeless parameters are copied inside the procedure. The use of this pragma must conform with the above pragma `t_l_size`, as it is not possible to copy the parameter if the size is not passed. The default setting is off.

```
var_str => on | off
```

This pragma is available under Pascal only. It controls whether the compiler checks that the actual parameter matches the formal parameter. The pragma only affects var string parameters. This pragma is useful for the conversion of Turbo Pascal programs, which has the same feature. The default setting is off.

```
inline_max => Number
```

This pragma controls the largest function which is inlined. The default setting is 12, which corresponds to the minimum code size for most programs. A larger value increases the code size and may accelerate code execution.

The pragma takes effect for each call, so a function may be called in different ways at different places.

Note: Functions are not inlined if the body has not been compiled before the call.

```
standard_float => on | off
```

This pragma is set on only when using the stack models. It must be used consistently throughout a program. When `standard_float=>on` is set, the compiler uses a stack model for the 8087, which means that the 8087 stack is empty after each statement. The default is on for stack calling convention, and off for the jpi calling convention.

```
standard_conv => on | off
```

This pragma is normally set together with the `standard_float` pragma above. The effect on C and C++ programs is the same as the `c_conv` pragma. For Modula and Pascal, there is no effect. The default is on for stack calling convention, and off for jpi.

```
opt_var_arg => on | off
```

This pragma controls whether optimized entry sequences are generated for procedures with variable parameter lists. The default is off for stack calling convention, and for overlay and DOS dynalink memory models. For other models in the jpi calling convention, the default is on.

Data Pragas

Pragas with the class name data affect data segmentation, data pointers and all aspects of data layout. The current settings of the data pragmas at the point of a variable's declaration will affect the way in which it is accessed.

```
seg_name => identifier
```

The pragma `data(seg_name=>xxx)` specifies that the compiler should place global initialized data objects in a segment named `xxx_DATA`, and global uninitialized data objects in a segment named `xxx_BSS`. These both have group name `xxx` and are in the `FAR_DATA` class. If the size of a data object is larger than the global data threshold, the compiler places the object in a separate segment.

The following example makes the names of the default segments `MYDATA_DATA` and `MYDATA_BSS`. These segments are in group `MYDATA` and have class `FAR_DATA`:

```
data(seg_name => MYDATA)
```

You can also specify `null`, to indicate the names `_BSS` and `_DATA`. The default value is `null` in all models except for extra large and multi-thread. For example:

```
data(seg_name => null)
far_ext => on | off
```

This pragma is available under C and C++ only. When on, the code generator does not assume that external variables are in the segment specified by the segment pragma. The pragma defaults to off in all models except for the extra large and multi-thread models. For example:

```
#pragma(seg_name=>MYDATA, far_ext=>off)
```

makes the name of the default segments `MYDATA_DATA` and `MYDATA_BSS` in group `MYDATA`. The compiler assumes external data objects to be in the same segment.

The default setting is on for overlay, dynalink, extra large and multi-thread models, otherwise off.

```
c_far_ext => on | off
```

This pragma is available under Modula-2 and Pascal only. When on, the code generator does not assume that external variables are in the segment specified by the segment pragma. The pragma defaults to off in all memory models. For example:

```
(*# data(seg_name => MYDATA,
        far_ext  => off )
*)
```

makes the name of the default segments MYDATA_DATA and MYDATA_BSS in group MYDATA. This pragma is not particularly useful in Modula-2 except for interfacing to C.

```
near_ptr => on | off
```

Specifies whether data pointers are near or far. This pragma also affects pointers generated by the & operator and by implicit array to pointer conversions in C and C++. For example:

```
data(near_ptr => on)
```

makes data pointers near, regardless of the model used.

near_ptr is on by default in small and medium models, but off for other models.

```
volatile => on | off
```

Variables declared when this pragma is set to on are considered to be volatile, and will always be kept in memory, rather than being kept in registers across statements.

The default setting is off. This pragma is not allowed in a project file, and is not available for C and C++, where the volatile keyword should be used.

```
volatile_variant => on | off
```

This pragma is available under Modula-2 and Pascal only. The effect is as for volatile above, but applies to variables of variant record types only. The default setting is off.

```
ext_record => on | off
```

This pragma is available under Modula-2 and Pascal only. Normally TopSpeed does not allow two fields in different alternatives of a variant record to have the same name. Using this pragma:

```
(*# data( ext_record => on ) *)
```

will allow you to use the same name in different alternatives, if the fields are located at the same offset in the variant record and they have the same data type.

The default setting is off.

```
var_enum_size => on | off
```

This pragma is available under Modula-2 and Pascal only. Enumeration constants with less than 256 alternative values are normally stored in one byte. Switching this option off:

```
(*# data( var_enum_size => off) *)
```

will force the compiler to store them as two-byte quantities. This is particularly useful for interfacing to third-party libraries and operating system calls that expect a word value. Without this pragma the enumeration would be byte rather than word size.

The default setting is on.

```
stack_size => Number
```

Specifies the size of the stack. You must place this pragma in the file containing the main function. If the stack size cannot be set to the specified size, the compiler uses the largest possible size. The default size is 16K bytes. For example:

```
#pragma data(stack_size => 0x6000)
main()
{
    /* statements */
}
```

makes the size of the run-time stack 0x6000 bytes (24K).

```
heap_size => Number
```

The pragma `heap_size` specifies the maximum size of the near heap. The near heap is the pool of free memory stored in the default data segment (DGROUP) which you can access with a near pointer.

In small and medium models, the standard heap functions operate on the near heap. Other heap functions operate on the near heap in all memory models. In compact, large, extra large and multi-thread models, the linker only makes a near heap if the program calls a near heap function.

The default near heap size (0xFFFF) occupies all the free space in the data segment after the allocation of stack and static data. If it is necessary to keep memory usage to a minimum, for example, when spawning other processes, set the maximum near heap size to the desired number of bytes. For example:

```
data(heap_size=>1000,stack_size=>2000)
```

sets the heap to 1000 bytes and stack to 2000 bytes.

Under DOS, the far heap size depends on the total memory available to a process when loaded. The heap size will be equal to the size of the memory block at load time, minus the total size of all code, static data, stack and near heap belonging to the process.

Under OS/2, the far heap size depends on the system memory configuration and available disk space for swapping.

Under Presentation Manager, the heap size pragma only controls the library near heap size; it does not control the size of the PM near heap.

Under Microsoft Windows, the heap size pragma has no effect.

```
ds_dynamic => on | off
```

This pragma controls the use of the DS register when its contents reference a shared, or potentially shared, memory segment. This is of particular importance when using the OS/2 inter-process communication mechanism or making DOS extra-large or multi-thread model programs.

One process communicates with another by passing the segment selector for a message block to the receiving process. After the message has been received the memory used is returned to the operating system heap. Thus the segment selector becomes invalid. Note that this mechanism applies to *processes*, not *threads*.

In order that the sending process does not attempt to use such selectors, the following pragma should be used:

```
data( ds_dynamic => off )
```

The default setting for all models is off.

```
share_global => on | off
```

This pragma is for OS/2 only. This causes the data segment to be shared among different OS/2 processes. This pragma can be used to allow different programs to communicate using shared variables. The default setting is off.

```
packed => on | off
```

This pragma is available under Modula-2 only. It controls whether record fields are packed at bit level. The default setting is off.

```
const_in_code => on | off
```

This pragma controls whether constants are put into a code or data segment. The default setting is on for extra large, multi-thread, and OS/2 dynalink models, and off for small, medium, compact, large, overlay, and DOS dynalink models.

```
ss_in_dgroup => on | off
```

This pragma specifies whether the stack is to be allocated in the default data segment (DGROUP). This is true in the small and medium models, but in the other models the stack is allocated in a segment of its own. In the larger models, you could want the SS register to equal the DS register when using Presentation Manager, or when linking with third party libraries. You need not specify this pragma under Presentation Manager or Microsoft Windows.

```
class_hierarchy => on | off
```

This pragma is available under Modula-2 and Pascal only. It controls whether information about class hierarchies is included in the class descriptor (method table). The information is used by the IS operator and type guards with check on. The default setting is on.

```
cpp_compatible_class => on | off
```

This pragma is available under Modula-2 and Pascal only. It controls whether the compiler includes extra information in class descriptors to provide compatibility with C++. The default setting is off.

```
compatible_class => on | off
```

This pragma is available under C++ only. It controls whether the compiler includes the correct information in class descriptors to provide compatibility with Modula-2 and Pascal. The default setting is off.

```
threshold => Number
```

This pragma sets the global data threshold. This determines at what size a data object is placed in a segment of its own. The default setting is model dependent:

Model	Threshold (bytes)
small	Not meaningful
compact	32767
medium	Not meaningful
large	32767
xlarge	65535
mthread	65535
overlay	65535
dynalink	65535

```
const_assign => on | off
```

This pragma is available under Modula-2 and Pascal only. It controls whether it is possible to assign to a structured constant. The default setting is off.

Warning: Under OS/2 and Windows 3, if `const_in_code` is on, assignments to constants will result in protection violations.

Check Pragmas

Pragmas with the class name check control run-time error checking. These can help you to locate erroneous program logic, but at the expense of slower execution. All these pragmas default to off, but can be set using the Project (C) Runtime Checks menu.

When a run-time check detects an error, the default action is to terminate the process and create the file ERRORINF.\$\$\$\$. Under the TopSpeed Environment, you can then use Utilities (E) Find run-time error to find the cause of the problem.

This behavior may be modified by installing a user defined handler. The files RTCHECK.H, RTCHECK.DEF and RTCHECK.ITS declare the procedure variables associated with the run-time checking error system and the associated error codes.

The user error-handler will be passed an error code, an IP value and a CS value, locating and identifying the error. If the handler returns a non zero value, execution will continue, otherwise the process will terminate.

Warning: It may not be safe to continue execution, especially under DOS.

```
stack => on | off
```

When on, the run-time system checks that your program does not run out of stack space. You can increase the size of the stack using the data(stack_size) pragma.

```
nil_ptr => on | off
```

When on, the run-time system checks for any dereference of NULL or NIL pointers.

```
range => on | off
```

This pragma is only available under Modula-2 and Pascal. When on, a range check is performed whenever a value is assigned to a variable of subrange or enumerated type. In addition, compile-time values are checked to ensure that they are in the range of their type.

```
overflow => on | off
```

This pragma is only available under Modula-2 and Pascal. When on, the run-time system checks that numeric values do not go out of range.

```
index => on | off
```

When on, the run-time system checks for the use of an array index larger than the array size.

```
guard => on | off
```

This pragma is available for Modula-2 and Pascal only. It controls whether checks are performed on the checked-guard operator.

```
field => on | off
```

This pragma is available for Pascal only. It controls whether checks are performed on access to record fields in records with tagfields.

The expr Pragma

There is a single pragma in the class expr:

```
promote => on | off
```

This pragma is available in Pascal source files only. It controls the manner in which expressions are promoted. The default is on.

For details of expression promotion in Pascal, see the “*TopSpeed Pascal Language Reference Manual*”.

Name Pragas

Pragas with the class name control aspects of linkage naming. However, the C++ programmer should also be familiar with C++ name mangling and extern declarations.

```
upper_case => on | off
```

This pragma is available in C and C++ only. It specifies whether public names should be converted to upper case. You would use this when interfacing to Pascal, or to third party C libraries. The default setting is off.

```
prefix => (none | modula | c | os2_lib | windows)
```

This pragma is available under Modula-2 and Pascal only (under C and C++ the syntax is slightly different Æ see below).

The name(prefix) pragma specifies the prefix and case of the public names that the compiler uses. The public names are names for non-static procedures and external data objects. By default, TopSpeed Modula-2 and Pascal prefix all external names with the name of the module followed by an ‘@’ for data and a ‘\$’ for procedures. You will need to use this pragma to interface to TopSpeed C.

The prefix pragma specifies which prefix scheme to use:

modula	use the TopSpeed Modula-2 naming convention of prefixing all external names with the name of the module and an ‘@’ or a ‘\$’.
none	puts no prefix on external names.

<code>c</code>	use the C naming convention (adds an underbar, ‘_’ to all external names).
<code>os2_lib</code>	use the OS/2 library standard (prefix all external names with the module name).
<code>windows</code> <code>prefix => string</code>	use the Microsoft Windows external naming convention.

This pragma is available under C and C++ only (under Modula-2 and Pascal, the syntax is slightly different – see above).

The value is a string specifying the prefix to all public names. An empty string specifies no prefix. The default prefix is an underbar.

If you wish to interface to TopSpeed Modula-2 or Pascal, you must use this pragma to specify the module prefix with a dollar (\$) suffix. For example, to use the `WrCard` procedure from module `IO`:

```
#pragma name(prefix => "IO$")
void WrCard(unsigned, int);
```

The default setting is language-dependent. The Project System does not set a default value for this macro.

Optimize Pragas

Pragas with the class name `optimize` control optimizations performed by the TopSpeed code generator. By default, all optimizations are enabled. Turning off an optimization will result in poorer code quality, and is unlikely to have a significant impact on compile times.

```
cse => on | off
```

When on, the compiler minimizes evaluation of complete expressions by keeping partial results in a temporary register. The default setting is on.

This option may be set using the Project Optimization (E) Common subexpressions menu command.

```
const => on | off
```

When on, the compiler will hold frequently used constants in registers to produce faster code. The default setting is on.

This option may be set using the Project Optimization Constant optimization menu command.

```
speed => on | off
```

When on, TopSpeed tries to make the code run as fast as possible without regard for the code size. When off, TopSpeed tries to make the code as small as possible.

A good example of the difference between optimizing for speed and optimizing for space is the use of a for loop. When optimizing for speed, the compiler might use nop instructions to place jump labels inside the for loop on even boundaries. The 80x86 architecture makes this much quicker than odd boundaries, but each nop adds another byte to the code size. This means that when optimizing for space, the compiler eliminates the extra nop instructions. The default setting is on.

This option may be set using the Project Optimization (T) Optimize for speed menu command.

```
stk_frame => on | off
```

When on, the compiler will only make stack frames where required, thus eliminating the need to set up the BP register. This optimization can only be made when all parameters and local variables for a function can be held in machine registers.

When off, the compiler always sets up the BP register, thus allowing a complete activation stack listing while debugging.

The default setting is on.

This option may be set using the Project Optimization (F) Stack frame menu command.

```
regass => on | off
```

When on, the compiler spends time finding the best allocation of registers for variables. This results in fast and tight code but slower compilation. The default setting is on.

This option may be set using the Project Optimization Register usage menu command.

```
peep_hole => on | off
```

When on, the compiler performs a variety of machine-code translations, generating smaller and faster code. The default setting is on.

This option may be set using the Project Optimization options (H) Peephole menu command.

```
jump => on | off
```

When on, the compiler will rearrange loops to eliminate as many jumps as possible, thus generating faster code. The default setting is on.

This option may be set using the Project Optimization Jump optimization menu command.

```
loop => on | off
```

When on, the compiler uses the loop depth when eliminating common sub-expressions and performing jump optimizations. The result of this optimization is faster, but potentially larger, code. The default is on.

This option may be set using the Project Optimization Loop optimization menu command.

```
alias => on | off
```

When on, this allows the compiler to assume that variables in a function will not also be used indirectly via a pointer in the same function. This assumption is not strictly allowed in ANSI C but is correct for all meaningful programs. The default setting is on.

This option may be set through the Project Optimization Alias optimization menu command.

```
cpu => 86 | 286 | 386 | 486
```

This controls the instructions used by the code generator, by declaring the processor to be used. If the target operating system is OS/2, the default is `cpu=>286`, otherwise the default is `cpu=>86`.

This option may be set using the Project Optimization Processor menu command.

```
copro => emu | 87 | 287 | 387
```

This controls what instructions the code generator may use by declaring which co-processor will be used. `copro=>emu` means that the 8087 will be emulated (if it is not detected at run-time). The default is `copro=>emu`, which gives maximum compatibility.

This option may be set using the Project Optimization (Q) Coprocessor menu command.

```
Debug Pragmas
```

Pragmas with the class name debug control the amount of additional information produced by the code-generator to assist debugging of programs.

```
vid => off | min | full
```

When full, the compiler places information for the TopSpeed VID debugger into a .DBD file. Use this option when debugging your program with VID.

Note: This disables the register usage and stack frame optimizations, allowing full access to variables within VID. All local variables are treated as volatile, to ensure that their values are not held in registers across statements, thus ensuring that VID can access their values at all times.

When `min`, the compiler performs the optimizations described above, and does not treat local variables as volatile. VID can still be used, but cannot reference local variables and some stack frames.

When `off`, the compiler generates no VID information, thus speeding compilation, generating the best possible code, and saving disk space.

```
The default setting is off.  
proc_trace => on | off
```

This pragma is available under Modula-2 and Pascal only. When this pragma is on, the compiler generates instructions to call the procedures `EnterProc` and `ExitProc` on, respectively, entering and exiting every procedure. These procedures can then perform any procedure tracing you may require.

Warning: You should ensure that this pragma is off for the `EnterProc` and `ExitProc` procedures themselves, otherwise infinite recursion will occur and your program will undoubtedly crash.

The two procedures must be *visible* to the module in which `proc_trace` is set on. This means that the module itself must define the procedures `ExitProc` and `EnterProc` or the module must specifically import them using an unqualified import.

The default setting is off.

```
line_num => on | off
```

This pragma causes the compiler to generate non-VID line number information for debuggers such as `symdeb`. This information is stored in object files and printed in the map file. The default setting is off.

```
public => on | off
```

This causes private objects to be made public to facilitate the use of debuggers such as `symdeb`. It may cause duplicated public warnings at link-time in languages such as C and C++ which do not have a modular structure. These warnings may be safely ignored, although it is recommended that such functions should be renamed to avoid possible confusion. The default setting is off.

Module Pragas

Pragas with the class name module control options that apply to an entire source file or module. These pragmas should be specified at the top of any source files to which they apply, or in the project file.

```
init_code => on | off
```

This pragma is available under Modula-2 and Pascal only. When on, it implies that the module contains initialization code to be run when the program is loaded and before the main module is executed. Switching the

option off is useful for modules written in other languages, as it will stop the linker warning of undefined symbols:

```
(*# module( init_code => off ) *)
```

Warning: If an implementation module sets this pragma off, then there is a knock-on effect, i.e., all imported modules must also have init_code set to off.

The default setting is on.

```
implementation => on | off
```

This pragma is available under Modula-2 and Pascal only. It specifies whether or not a definition file (.DEF or .ITF) has a corresponding object file. It should be turned off if the definition file defines interfaces to routines in a different language, to prevent the Project System from attempting to remake the corresponding object file. The default is on.

This pragma can also be used in the implementation part of a module, before any module source code. In this case it overrides the default naming of the associated object file. Normally the name of the .OBJ file corresponding to a module is taken from the module name. When this pragma is set off, the object filename will be taken from the filename, not the module name.

```
smart_link => on | off
```

Setting this pragma to off disables the smart linking feature, to the extent that either all or none of the objects in each segment from a compilation will be included in a link. This may result in quicker linking, and also may allow other linkers (such as Microsoft) to be used. (There are many potential problems with trying to use a non-jpi linker, and it is definitely not recommended). The default setting is on.

This option may be set using the Project Optimization Smart linking menu command.

```
turbo_comp => on | off
```

This pragma is available under Pascal only, for use with Turbo to TopSpeed converter. See the “*TopSpeed Pascal Language Reference Manual*”. The default setting is off.

```
init_prio => Number
```

This pragma is available under C++ only. It defines a priority for the initialization code for static objects. Normally the initialization order is undefined between files, but this pragma allows you to control the initialization order in that files with higher priority is initialized before modules with lower priority. The number must be a value between 0 and 32. The default value is 16, and the C++ library uses values between 25 and 32 (it is therefore not recommended to use values in this range, otherwise part of the library may not have initialized before user code is executed).

Option Pragmas

Pragmas with the class name option control language-dependent options, such as TopSpeed extensions.

```
ansi => on | off
```

This pragma is available under C and C++ only. If it is set to on, ANSI keywords only are allowed. The default setting is off.

This option may be set using the Project Language options ANSI keywords menu command.

```
iso => on | off
```

This pragma is available under Pascal only. If it is set to on, no TopSpeed language extensions are allowed, and the compiler operates in ISO standard mode. The default setting is off. In order for the compiler to behave in a fully ISO conformant way, this option must be specified in the project file or on the command line, rather than in the source code.

```
lang_ext => on | off
```

This pragma is available under C and C++ only. The following constructs are not valid under ANSI C, but are included in TopSpeed C and C++ when this pragma is not set on:

- A type cast yields an lvalue if the operand is an lvalue.
- Functions can be initialized with binary machine code.
- The relational operators (>,>=,<=,<) allow the operators to be a mixture of integer and pointer operands.
- Bitfields in C can have type char and unsigned char.
- Relative pointers.

The default setting is on.

This option may be set using the Project Language options (E) Language extensions menu command.

```
nest_cmt => on | off
```

This pragma is available under C and C++ only. When on, you can nest comments without causing an error message. For example:

```
/* This is a test comment  
/* This is a nested comment */  
*/
```

When off, nested comments cause an error message. The default is off, allowing the compiler to trap unterminated comments more easily and make it conform with ANSI C.

This option may be set using the Project Language options Nested comments menu command.

```
uns_char => on | off
```

This pragma is available under C and C++ only. When on, types declared as char lie between 0 and 255. When off, values declared as char lie between Æ127 and 128. The default setting is off.

This option may be set using the Project Language options (J) Default char unsigned menu command.

```
pre_proc => on | off
```

This pragma is available under C and C++ only. When on, the compiler produces preprocessor output in a file with the same name but with extension .i. This output file makes it easy to debug the result of macro expansions. The default setting is off.

```
incl_cmt => on | off
```

This pragma is available under C and C++ only. When on, comments are preserved in preprocessor output. The default setting is off.

This pragma has no effect unless pre_proc is on.

```
min_line => on | off
```

This pragma is available under C and C++ only. When on, the preprocessor minimizes the number of blank lines in output. The default setting is on.

This pragma has no effect unless pre_proc is on.

```
prototypes => on | off
```

This pragma is available for C only. When on, the compiler writes prototype declarations for function definitions to a new file with the source filename and the extension .PT. This file can then be included into source files that call these functions via the #include directive. In this way, you have the advantage of argument type checking for function calls, which allows safer programming and better code generation. For example:

```
struct ST {int i;};

int t1(const int ci, int *vpci)
{
    /* statements */
}
int t2(const char acc[2], const long aac1[3][4])
{
    /* statements */
}
int t3(unsigned (*paaau)[5][6][7], int *api[3])
{
    /* statements */
}
int t4(char (*pfc)(char *pc, const int *pci))
{
    /* statements */
}
int t5(struct ST s, const struct ST cs)
{
    /* statements */
}
```

generates the definition file below:

```
int t1(const int ci, int *vpci);
int t2(const char *acc, const long (*aac1)[4]);
int t3(unsigned (*paaau)[5][6][7], int **api);
int t4(char (*pfc)(char *pc, const int *pci));
int t5(struct ST s, const struct ST cs);
```

The compiler does not retain typedef names in the generated prototypes. Instead, it uses the corresponding base types. For example:

```
typedef int *INTEGERPTR;

int AFunc(INTEGERPTR APtr)
{
    /* statements */
}
```

will generate the prototype:

```
int AFunc(int *APtr);
```

Note: Enumeration types and structure types must have tags, otherwise the compiler inserts question marks (?) at the tag position.

The default setting is off.

This option may be set using the Project Language options (Z) Generate prototypes menu command.

```
bit_opr => on | off
```

This pragma is available under Modula-2 only. It allows bitwise operations on cardinals:

```
( a AND/OR b, NOT a ).
```

The default setting is off.

This option may be set using the Project Language options Bitwise boolean operators menu command.

Warn Pragmas

Pragmas with the class name warn control the generation of compiler warnings. These pragmas are only available under C and C++.

The warnings given by TopSpeed C and C++ help you to check, as far as possible, common coding mistakes. Since no compiler can determine your intentions, you may get warnings even if your code is correct. Your code may generate some warnings more than others, so TopSpeed allows you to customize which warning checks are performed.

You can set each of the warning options to on, off or err. When on, TopSpeed checks the code for that warning and reports the problem, but the problem

does not stop the compilation or linking. When off, TopSpeed ignores the warning. When err, TopSpeed checks the code for the warning, reports the problem, and does not allow linking until you have fixed the problem.

Note: TopSpeed C and C++ check the code and produce a warning for a good reason. Indeed, to utilize your non-ANSI C code, TopSpeed C uses a minimal set of warning messages by default. You should, therefore, think twice before turning off any of the default warning messages. We advise that you keep all the warnings either on or err.

All warn class pragmas may be set using the Project Language options (V) Ansi violation warnings, (T) Type check warnings, (X) Possible errors and (C) C++ warnings menus.

```
wall => on | off | err
```

This pragma affects the settings of all the warnings. If set to on or err, all warnings will be enabled.

```
wpcv => on | off | err
```

Pointer conversion. When on or err, the compiler checks for a conversion between two incompatible pointer types, or between a pointer and an integral type. The default setting is on.

```
wdne => on | off | err
```

Declaration has no effect. When on or err, the compiler checks for a declaration that has no meaning, for example, long int;. A declaration should contain a variable declarator, a structure or union tag, or members of an enumeration. The default setting is on.

```
wsto => on | off | err
```

Storage class redeclared. When on or err, the compiler checks that you have not declared the same variable differently within your program. For example:

```
int x;          /* External linkage */  
static int x; /* Internal linkage */
```

The static storage class always takes preference. The default setting is on.

```
wtxt => on | off | err
```

Unexpected text in preprocessor command. When on or err, the compiler checks for a new line character terminating a preprocessor command. The default setting is on.

```
wprg => on | off | err
```

Unknown pragma. When on or err, the compiler checks for foreign pragmas or mistakes in TopSpeed C pragmas. If you are only creating code using TopSpeed C or C++ pragmas, you should switch this warning to either on or err. The default setting is on.

```
wfnd => on | off | err
```

Function not declared. When on or err, the compiler checks for functions that have been called but not declared. If these functions occur, TopSpeed

assumes that the function is an extern function returning an int. The default setting is off.

```
wynd => on | off | err
```

Function prototype not declared. When on or err, the compiler checks whether a function has a prototype associated with it. Prototypes are important to TopSpeed, since it can not do much type checking without them. You should, therefore, declare prototypes for all functions. It is best to keep this warning on or err. The default setting is off.

```
wnre => on | off | err
```

No expression in return statement. When on or err, the compiler checks for a return value in a non-void function. You should keep this warning off if you are compiling some old style C code without prototypes. The default setting is off.

```
wnrv => on | off | err
```

No return value in function. When on or err, the compiler checks for a return statement in a non-void function. The default setting is off.

```
watr => on | off | err
```

Different const attributes. When on or err, the compiler checks whether a function that expects a pointer to a variable gets a pointer to a constant. The default setting is on.

```
wftn => on | off | err
```

Far to near pointer conversion. When on or err, the compiler checks for conversion of a 32-bit far pointer to a 16-bit near pointer. The default setting is on.

```
wntf => on | off | err
```

Near to far pointer conversion. When on or err, the compiler checks for conversion of a 16-bit near pointer to a 32-bit far pointer. The default setting is on.

```
wubd => on | off | err
```

Possible use of variable before assignment. When on or err, the compiler checks that you have used a local variable before you have given it a value. TopSpeed checks this warning with a simple scan through the function, which can cause gotos and the like to generate false warnings. The default setting is on.

```
wpnu => on | off | err
```

Parameter never used in function. When on or err, the compiler checks for a parameter that the code never uses, so declaration of dummy parameters generates warnings. The default setting is off.

```
wdnu => on | off | err
```

Variable declared but never used. When on or err, the compiler checks whether a local variable has been declared but never used in the function. The default setting is on.

```
wcne => on | off | err
```

Code has no effect. When on or err, the compiler checks statements and the left operand in a comma expression to see if they have no effect. The default setting is on. For example:

```
x + y;      /* expression has no effect */
f, x;       /* left operand has no effect */
wcld => on | off | err
```

Conversion may lose significant digits. When on or err, the compiler checks for a conversion from long or unsigned long to int or unsigned int. The default setting is on.

```
wait => on | off | err
```

Assignment in test expression. When on or err, the compiler checks for a possible mistyping of the C equality (==) operator. The equality operator contains two =. For example:

```
if (x=y) printf("X equals Y"); /*is a mistake*/
```

The default setting is on.

```
wetb => on | off | err
```

Value of escape sequence is too large. When on or err, the compiler checks that an escape sequence is in the range 0 to 255. The default setting is on.

```
wcor => on | off | err
```

Value of constant is out of range. When on or err, the compiler checks whether an integer constant is in the range of an unsigned long, or a floating point constant is in the range of a long double. The default setting is on.

```
wclt => on | off | err
```

Constant is long. When on or err, the compiler checks for an integral constant that has type long because of its value but does not have an L suffix. The default setting is off.

```
wral => on | off | err
```

Returns address of local variable. When on or err, checks for a return statement that returns the address of a local variable. This causes a problem because C reclaims the variable storage on completion of the function. The pointer, therefore, points at undefined data. The default setting is on.

```
wpin => on | off | err
```

Default type promotion on parameter. When on or err, the compiler compares the declaration of a parameter in an old-style function definition with the prototype for incompatible argument promotions. For example:

```
int func(char);
        /* parameter declared as char */

int func(IntegerByPromotion);
char IntegerByPromotion;
        /* INCOMPATIBLE */
{
        /* Statements */
}
```

Note: This is a violation of the ANSI C standard regarding compatible function declarations. The default setting is on.

wpic => on | off | err

Parameter list inconsistent with previous call. This warning is issued if a parameter declaration is incompatible with the corresponding parameter in a previous function declaration. The default setting is on.

wnid => on | off | err

Address for local variable not in DGROU. When on or err, the compiler checks that a local variable does not have its address taken in small model, when using the #pragma data(ss_in_dgroup => off). The default setting is on.

wrfp => on | off | err

Function redeclared with fixed parameters. When on or err, the compiler checks for a prototype with a variable number of arguments, where the corresponding function definition specifies a fixed number of arguments. This will work in TopSpeed C, but it is a violation of the ANSI C rules for compatible function declarations, and therefore, not portable. The default setting is on.

wvnu => on | off | err

Local variable never used. When on or err, the compiler checks whether you declare a local variable and assign it a value but never use it. The default setting is on.

wovr => on | off | err

Overflow in constant expression. This warning is issued when a constant integer expression overflows. The default setting is on.

wacc => on | off | err

Default access specifier used for base class. This warning is issued if a base class specification does not have an access specifier and the default access is used (i.e. public for a struct and private for a class). The default setting is on.

wdel => on | off | err

Expression in delete[] is obsolete. This warning is issued if an expression is specified in the square brackets of a delete expression. The expression is ignored. This is obsolete C++ usage. The default setting is on.

wovl => on | off | err

Keyword ‘overload’ is not required. This warning is issued if the keyword overload is specified in C++. The use of this keyword is obsolete C++ usage. The default setting is on.

```
wcic => on | off | err
```

Constant in code segment requires initialization. This warning is issued if a constant placed in the code segment requires run-time initialization, as may be the case for an object declared const in C++, whose initializer is an expression, when the const_in_code pragma is set on. This situation will lead to a protection violation in OS/2 and Windows 3 protected mode applications, so const_in_code should be set off if this warning is encountered. The default setting is on.

```
wcrt => on | off | err
```

Class definition as function return type, missing ‘;’ after ‘}’?

This warning is issued if a class is defined in a function return type specification. Such a construct is legal, but unusual, and frequently results from omitting a semicolon between a class definition and the following function declaration. The default setting is on.

The project Pragma

A pragma with the class name project is used to pass information from a compilation to the Project System. The value of the pragma should be a string, which is then stored in the object file for the Project System. Whenever an object file is added to the link list, the text specified using this pragma is executed as a Project System command. For example, if a header file includes the line:

```
#pragma project("#set myflag=on")
```

then whenever a source file that includes this header file is included in a project, the Project System macro myflag will be set. This might be used for processing later in the project file.

This pragma may only appear in source files, not in a project file.

The save and restore Pragas

The save pragma saves the entire pragma state, so you can later restore it with a restore pragma. The save and restore pragmas work in a stack-like manner, thus allowing you to nest them. For example:

```
/*
  save the pragma state and enable the
  interrupt convention
*/
#pragma save
#pragma call(interrupt => on)
/* interrupt functions are specified here */
#pragma restore
```

There is no limit on the number of saves, except the amount of memory available. These pragmas may be used in source files or in a project file.

Link Pragmas

The link pragma and the linkfirst pragma are used to add files to the link list (see the section on '*The Link List*' in Chapter : '*The Project System*').

```
link( <filename> {,<filename> } )
linkfirst( <filename> )
```

These pragmas may be specified in a project file, and causes the nominated files to be added immediately to the link list. For example:

```
#pragma link( file1.obj, file2.obj, file3.lib )
#pragma linkfirst (initexe.obj)
```

If no extension is given .obj is assumed. Files specified using #pragma link are added to the end of the link list (unless already present). A file specified using #pragma linkfirst is linked before the link list. Only one file may be specified for each link using #pragma linkfirst.

In addition, the link pragma may be specified in a C or C++ source file, in which case the nominated files will be added to the link list when an autocompile command is executed in the Project System, if any files already on the link list had this pragma specified. When used in a source file, the filename may not specify a path or extension.

Link_option Pragmas

Pragmas with the class name link_option are used to specify linker options. These pragmas may only occur in project files.

```
map => on | off
```

Controls whether a map file is generated with information about segment sizes and publics etc. The default is to create a map file.

This option may be set using the Project Project options Map file menu command.

```
case => on | off
```

This pragma controls whether the linker treats upper/lower case as significant when linking. The default is case=>on, unless any Pascal source files have been included in the project, in which case the default is case=>off.

This option may be set using the Project Project options Case sensitive link menu command.

```
pack => on | off
```

This pragma controls whether segments are packed together. The default is `pack=>off` for Windows executables, and for DOS executables in overlay model, and `pack=>on` otherwise.

```
export => on | off
```

This option is required if a .EXE file has exports (for example a Windows program). It is not required for .DLL files (since these are assumed to always have exports). The default is on for Windows programs, otherwise off.

```
windows => on | off
```

This option is needed when linking a Windows executable. The default is on when `#system win` is specified, and off otherwise.

```
oldexe => on | off
```

This option specifies whether a new format executable file (as used under OS/2 or Windows, and in overlay model for DOS executables) or an old format (as used for standard DOS executables) is required. The default is set appropriately by the Project System depending on the values specified by the `#system` and `#model` commands.

```
pm => on | off
```

When on, an OS/2 EXE header will indicate that the program uses the API provided by Presentation Manager. This is the equivalent of the module definition application type `WINDOWAPI`.

If both `pm` and `non_pm` are set to on, the program will run in a PM window, or separate screen group. An application is of this type if it uses the subset of OS/2 VIO, KBD, and MOU calls common to both PM and the character mode API. This is the equivalent of the module definition application type `WINDOWCOMPAT`.

The default setting is off.

```
non_pm => on | off
```

When set to on, the OS/2 EXE header indicates that the program does not run in a Presentation Manager window. This forces the program to use a separate screen group, but allows you to use all of the OS/2 VIO, KBD, and MOU functions. This is the equivalent of the module definition application type `NOTWINDOWCOMPAT`.

```
overlay => on | off
```

This option is required when linking a DOS executable in overlay model. The Project System will set this option automatically when `#model overlay` is specified.

```
stub => <filename>
```

This specifies the name of the stub file to be used for an OS/2 or Windows format executable. If no stub program is specified, the linker inserts the default stub.

```
decode => on | off
```

This indicates whether the linker should produce decoded names in the MAP file, as well as their public symbols. The option is set to on if any C++ source files are included in a project, otherwise off.

```
shift => num
```

This specifies the segment alignment shift count for new-format executables, (for example, OS/2 or Microsoft Windows programs, or overlay model programs). The default is 4, indicating that segments are aligned on 16-byte boundaries.

The define Pragma

A pragma whose class name is define is used to define a conditional compilation symbol for subsequent compilations. This pragma may only be used in project files.

A define pragma takes the form:

```
#pragma define(ident=>value)
```

where ident names the symbol to be defined, and value specifies the value it is to be given.

For Modula-2 and Pascal, the given identifier is defined as a boolean constant with value TRUE if the value on was specified, otherwise FALSE.

For C and C++, the given identifier is defined as a macro. If the value on is specified, the macro is defined to the value 1. If the value off is specified, the macro is not defined. Any other value will cause the identifier to be defined as a macro expanding to the given value. Only a single C or C++ token may be specified, or the compiler will report an error. To define a macro where the value is a string literal, use a command of the form #pragma define (name => “fred”).

CHAPTER 5

Conditional Compilation

TopSpeed provides system wide, language independent *conditional compilation*. This means that you can place special commands in your source file that will cause different parts of the source to be compiled depending on some simple boolean values.

The biggest use for this feature is to produce programs that can be compiled under a number of different memory models from a single source file.

Modula-2 Syntax

The syntax for conditional compilation is as follows:

ConditionalComp ::= *Test SourceText* ‘(*%E*)’

Test ::= { ‘(*%T’ *Identifier* ‘*)’ |
 ‘(*%F’ *Identifier* ‘*)’ }

The *source text* consists of Modula-2 statements, declarations or compiler pragmas.

Note: The *identifier* must be an unqualified BOOLEAN constant that has been assigned a value before the occurrence of the conditional compilation directives. As explained below, there are a number of predefined identifiers that take values depending on the options set in the TopSpeed Project System or command-line pragmas. These are always defined for every compilation.

The (*%T form causes the source text to be compiled if the value of the identifier is TRUE. The (*%F form causes the source text to be compiled only if the value of the identifier is FALSE. For example:

```
MODULE Hello;
IMPORT IO;
CONST
  French   = FALSE;
  German   = TRUE;
  English  = FALSE;
BEGIN
  (*%T French *)
    IO.WrStr("Bonjour");
  (*%E)
  (*%T German *)
    IO.WrStr("Guten Tag");
  (*%E)
  (*%T English *)
    IO.WrStr("Good day");
  (*%E)
END Hello.
```

If you run this program, the compiler will only compile the lines between `(*%T German *)` and `(*%E)`. All other `IO.WrStr` calls are ignored.

Pascal Syntax

The syntax for conditional compilation is as follows:

ConditionalComp ::= *Test SourceText* `(*%E*)` |
Test SourceText `{%E}`

Test ::= { `(*%T` *Identifier* `*)` |
`(*%F` *Identifier* `*)` } |
{ `{%T` *Identifier* `}` |
`{%F` *Identifier* `}` }

The *source text* consists of Pascal statements, declarations or compiler pragmas.

Note: The *identifier* must be an unqualified BOOLEAN constant that has been assigned a value before the occurrence of the conditional compilation directives. As explained below, there are a number of predefined identifiers that take values depending on the options set in the TopSpeed Project System, or by the command-line pragmas. These are always defined for every compilation.

The `(*%T` form causes the source text to be compiled if the value of the identifier is true. The `(*%F` form causes the source text to be compiled only if the value of the identifier is FALSE.

For example:

```
program Hello(output);

const
  French = false;
  German = true;
  English = false;

begin
  (*%T French *)
    writeln('Bonjour');
  (*%E)

  (*%T German *)
    writeln('Guten Tag');
  (*%E)

  (*%T English *)
    writeln('Good day');
  (*%E)

end.
```

If you run this program, the compiler will only compile the lines between (*%T German *) and (*%E). All the other `writeln` calls will be ignored.

C and C++ Syntax

Both the forms `#if` or `#ifdef` may be used. For example:

```
#ifdef _mthread
...
#endif

#if _mthread
...
#endif
```

Predefined Identifiers

Whenever you compile a program, the Project System predefines a number of identifiers, depending on the model and target operating system. In the case of C and C++, these identifiers are macros, and are either defined to the value 1 or left undefined as appropriate. These can therefore be tested using `#if` or `#ifdef` statements as preferred. In the case of Pascal and Modula-2, the identifiers are defined as boolean constants, with the value `TRUE` or `FALSE` as appropriate. The following identifiers are defined:

<code>_fcall</code>	calls and returns are far.
<code>_fptr</code>	pointers are 32-bit.
<code>_fdata</code>	the ds register not = dgroup on function entry.
<code>_mthread</code>	the program is multi-threaded.
<code>_jpicall</code>	parameters are passed in registers.
<code>_WINDOWS</code>	target system: Windows
<code>_DLL</code>	target system: DLL
<code>_WINDLL</code>	target system: Windows DLL
<code>_OVL</code>	target system: Overlay
<code>_OS2</code>	target system: OS/2
<code>__OS2__</code>	target system: OS/2
<code>__MSDOS_</code>	target system: MSDOS
<code>M_I86xM</code>	target is 8086, memory model indicated by x. x may be S, M, C, L, T, X, O or D.

Other conditional-compilation values may be set in the Project file using the `pragma define(identifier=>value)`. See the section on the ‘*The define Pragma*’

in Chapter : '*Pragmas*'. There are a number of major uses for conditional compilation:

- Including debugging code. You can conditionally include calls to debugging procedures in your program. You do not need to delete these lines when the project is complete Æ you simply set a constant to FALSE.
- Selecting different algorithms or procedures depending on the configuration of the memory model. Most TopSpeed libraries use this technique to keep down the number of source files. It also means that changes independent of the memory model only have to be done once.
- Producing different versions of your program Æ for example, a demonstration version with some features missing or with limits on file sizes.
- Producing different versions of your program dependent on target operating system.
- Handling memory model dependencies.

CHAPTER 6

Segment-based Overlays

Overview

TopSpeed's *Segment-based Overlay Management System* (OMS) allows you to write programs of virtually any size (up to 254 segments) which will run on any DOS machine. It achieves this by swapping code and data segments between the main memory of the machine and either a hard disk or EMS (Expanded) Memory. The facility does not require the machine to be run in protected mode.

Unlike the standard Microsoft DOS overlay system, TopSpeed OMS will free up main memory to make space for any segments which have to be swapped in, if necessary writing out changed data to disk or EMS to save it.

TopSpeed OMS also differs from standard overlay management systems in that it will operate entirely automatically. In a normal program, there is no requirement on the programmer to specify an overlay structure: all you have to do is specify a special "Overlay" memory model.

If you wish to control the overlay process in more detail, the facilities are described in the "*TopSpeed Advanced Programmer's Guide*", part of the TopSpeed Techkit[®].

Overlays and Multi-thread Programs

While the programmer is not in general obliged to take control of the overlay process, there is one case in which it is obligatory. This is when overlays are combined with multi-thread programs. The reason is that DOS restrictions make it impossible to guarantee that a segment is inactive under all conditions. Making multi-thread programs using overlays is described in the "*TopSpeed Advanced Programmer's Guide*".

OS/2

The overlay system is for DOS use only. Under OS/2 overlays are superfluous, since the system provides comprehensive memory management in a 4GB virtual address space. However OS/2 users can benefit significantly from creating their own *Dynamic Link Libraries* (see Chapter : '*Dynamic Link Libraries for OS/2*'), which facilitates code-sharing and more economical use of memory.

Compiling and Running an Overlayed Program

Using the overlay system is simplicity itself. All you need to do is specify a special “Overlay” memory model. This can be done in two ways from within the TopSpeed Environment:

- Edit the project file to include the statement
`#model overlay`
- Use the Project Memory model environment menu option to set the model.

The Project System will automatically compile and link your program into an overlayed .EXE file.

Files Used by the Overlay Linker

The only file, over and above the normal ones, which is required by the linker in order to create an overlayed program is called OVERLAY.EXP. The default redirection file will tell the linker where to find this file. You, therefore, need take no special steps except to ensure that this file is present and accessible through the redirection file.

Running an Overlayed Program

You need take no special steps to run an overlayed program. It may be called like an ordinary .EXE file. The only requirements to observe are:

- There must either be expanded memory and disk space available with sufficient room for swapping.
- The program must be resident on the disk drive from which it was loaded for the duration of its execution.

Design Considerations

Provided the above conditions are met, any overlayed program will always run. However, the design of the program can affect performance if there are a large number of segments making heavy demands on memory. You should try to design your program to avoid ‘thrashing’ (swapping much-used segments continually in and out of memory). Remember also that small segments will minimize granularity.

There are several factors to take into account:

- The program layout and its division into segments. This is controlled by the organization of the source code into units or by using the call(seg_name=><name>) pragma. This forces any code which follows into the segment

called *<name>*. It can be used both to break a compilation unit into parts, and to combine code from different compilation units.

- The use of interrupt handlers and manual overlay control requires the TopSpeed TechKit[®].

Program and Data Layout

The basic unit of memory used by the overlay system for swapping is the *segment*. Entire segments are swapped in and out. Therefore, the key to good design is to keep related parts of the program in the same segment. Thrashing will be less likely if the program spends most of its time moving around within a single segment, with less frequent inter-segment jumps. The same principles apply to data segments, which are demand-loaded.

The layout of a program in segments is determined in the following way. The default under the overlay model is to place each *compilation unit* (Modula-2 module, Pascal unit, C/C++ source file) in a single segment. A separate *data segment* is created for the static data declared in each compilation unit.

Points to Note

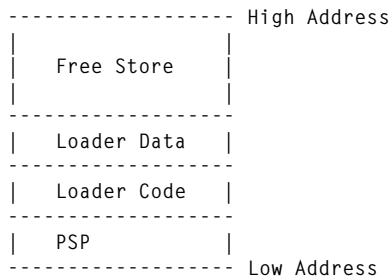
The basic principles of good segment layout are summarized below:

- Avoid large amounts of static preloaded data.
- Make all intra-segment calls near.
- Avoid using constants in code unless the segment containing constant data is resident.
- Small modules will reduce granularity.
- Do not override the default pragma settings `same_ds` or `ds_entry`.

The Default Memory Management Algorithm

The default overlay algorithm will be applied if you take no special action. It is useful to understand how this works, particularly if you intend to override it.

An overlay program looks like a normal .EXE program to DOS. When you create an overlay program, the linker places a special loader and its data segment at the start of the program (in low memory) immediately after the Program Segment Prefix. Any segments marked as PRELOAD are also loaded. The remaining free store is shared between the overlay manager and the far heap. Any library functions that return information on the amount of memory in the far heap will indicate the amount available for allocation for both overlay code and user data.



The remaining segments are then loaded as demanded.

The rules are:

- Any code segment will be loaded whenever a function call is executed which references the segment, if it is not already loaded.
- If dynamic data loading is enabled, a static data segment will be loaded whenever a code segment that refers to it is loaded.

If insufficient memory is available to satisfy a request either from user heap functions or from the overlay manager, inactive segments are unloaded until sufficient space has been made available. The algorithm for deciding which segment to unload is an LRU (Least Recently Used) algorithm.

If there is still insufficient memory, a user memory allocation request will fail. This may be handled in the normal manner for the language involved. If the request was from the overlay manager, the process will terminate. However, the loader may not be able to load the segments needed to execute the error handler, in which case the process will terminate immediately.

If present, expanded memory is used as a temporary store for code segments, and to fix frequently used segments.

Memory Available Functions

Functions that indicate the amount of memory in the far heap, such as `Storage.AVAILABLE` (Modula-2) and `farcoreleft`, `coreleft` (C) can give misleading results in overlay or dynalink models.

These functions return the number of bytes remaining unallocated at the moment of the call. If a following call to an allocation function fails, the overlay manager will unload inactive code and data segments increasing the number of free bytes. Therefore a simple call to one of these functions is not a reliable method of determining how much memory is available.

One solution is to call `UserFlush` before calling `coreleft` for example. This will give an accurate result but is not optimal. A better solution is to handle the error return from the allocation function explicitly:

Modula-2	Set Storage.Check to FALSE and check for NIL.(See the “ <i>TopSpeed Modula-2 Reference Manual</i> ”)
C & C++	Check for a NULL return value.
Pascal	Install a heap error handler returning 1. (See the “ <i>TopSpeed Pascal Library Reference Manual</i> ”)

Programming for the Overlay Model

The overlay memory model is a superset of the multi-thread model, which is in turn a superset of the extra large model. The characteristics of each of these models is relevant to the overlay model’s performance.

The extra large model differs from the large model in one major respect Æ it provides separate data and code segments for each compilable module, giving the programmer sufficient control over granularity to create an efficient overlay.

The multi-thread model provides extra resources required to run multi-thread programs under DOS or OS/2. However, for the reasons explained above, if multi-thread programs are using overlays they must take explicit control over the overlay procedure in order to ensure that a segment being referenced by one thread is not swapped out by another. See the “*TopSpeed Advanced Programmer’s Guide*” for details.

Overlay System API

A number of library functions are provided to allow some control over the overlay process. The full overlay API is described in the “*TopSpeed Advanced Programmer’s Guide*”. For the purposes of this manual, the most important such function is UserFlush, which unloads all inactive overlays and DLLs. This is defined as follows:

C/C++	void UserFlush(void); in overlay.h
Pascal	PROCEDURE UserFlush; in overlay.itf
Modula-2	PROCEDURE UserFlush(); in overlay.def

If any of the above functions are used, the Project System will ensure that the appropriate library is included in the link.

Run-time Errors

LOADER_ERROR_INVALID_ENTRY (Code 1)

Invalid Entry in executable file entry table. Indicates invalid file.

Action: Check project files and remake.

LOADER_ERROR_TOO_MANY_MODULES (Code 2)

Too many DLLs in project or loaded by LoadModule.

Action: Re-structure project and remake.

LOADER_ERROR_WRONG_MODULE_VERSION (Code 3)

Wrong version of DLL present.

Action: Remake.

LOADER_ERROR_MODULE_CORRUPT (Code 4)

DLL or executable file is corrupt.

Action: Remake.

LOADER_ERROR_STACK_CORRUPT (Code 5)

Stack chain used for unloading segments is corrupt.

Action: Check program logic.

LOADER_ERROR_MODULE_NOT_FOUND (Code 6)

Imported DLL cannot be found.

Action: Check file exists and remake if necessary.

LOADER_ERROR_FIXUP_INCORRECT (Code 7)

Internal fixup incorrect.

Action: Report to JPL.

LOADER_ERROR_EMS_ERROR (Code 8)

EMS error.

Action: Check if any other resident processes are using EMS and not restoring

page context.

LOADER_ERROR_NOT_DLL (Code 9)

File loaded was incorrect format.

Action: Remake.

LOADER_ERROR_STACK_TRACE (Code 10)

Debugging version error.

Action: report to JPI.

LOADER_ERROR_MODULE_INIT_FAILED (Code 11)

DLL initialization returned non-zero indicating failure.

Action: Check program logic.

LOADER_ERROR_FILE_NOT_FOUND (Code 12)

Imported DLL cannot be found.

Action: Check file exists on PATH and remake if necessary.

LOADER_ERROR_OUT_OF_MEMORY (Code 13)

Process out of memory.

LOADER_ERROR_FILE_READ (Code 14)

I/O error on file read.

LOADER_ERROR_FILE_WRITE (Code 15)

I/O error on file write, probably disk full.

LOADER_ERROR_STACK_OVERFLOW (Code 16)

Stack overflow in loader.

Action: increase stack size.

LOADER_ERROR_CODE_FIXUP (Code 17)

Illegal fixup to code segment.

Action: Check `const_in_code` setting and see guide to assembly programming.

LOADER_ERROR_TOO_MANY_SEGMENTS (Code 18)

Too many segments in module.

Action: reduce number of segments by grouping.

LOADER_ERROR_INVALID_LOAD (Code 19)

Internal error.

Action: Report to JPI.

LOADER_ERROR_CMEM_ERROR (Code 20)

Internal error.

Action: Report to JPI.

LOADER_ERROR_INVALID_IMPORT (Code 21)

Internal error.

Action: Report to JPI.

LOADER_ERROR_NO_ENTRYPOINTS (Code 22)

Internal error.

Action: Report to JPI.

LOADER_ERROR_ILLEGAL_TRANSFORM (Code 23)

Internal error.

Action: Report to JPI.

LOADER_ERROR_SWAP_FILE (Code 24)

Error reading/writing Swap file.

Action: Check free disk space.

LOADER_ERROR_ILLEGAL_ADDITIVE (Code 25)

Internal error.

Action: Report to JPL.

Limits and System Requirements

The new executable file format imposes a limit of 254 segments on the exported entry points in a module, and 333H on the number of entry points in any one module. Using the `link_option(pack=>on)` pragma reduces the total number of segments in a module, but may increase granularity.

There are also internal limits to the loaders capacity. The maximum number of active segments is 512.

There is no limit on total code size apart from the limit on the total number of segments. Therefore it would be possible to have a single executable file containing up to 16 MB of code.

Preloaded static data and code is limited by available memory.

The minimum efficient size for a segment is 128 bytes.

System Requirements

The loader uses interrupt 3FH. This may not be used by your process.

CHAPTER 7

Dynamic Link Libraries for OS/2

Dynamic Link Libraries (DLLs) are a major innovation brought about by OS/2. DLLs allow your applications to share common data and code which is incorporated into your program at load time, rather than link time. They have the same advantages as traditional linkers which are:

- They save disc resources, since it is no longer necessary for each executable file to contain its own copy of a functions.
- They make product updates much easier, since only specific DLLs need be updated instead of the entire program.
- They permit code to be shared at runtime, saving main memory.

DLLs are created very simply under TopSpeed, using the Project System and a special memory model known as the *dynalink model*. This is a superset of the multi-thread model.

The Advantages of Dynamic Linking

For most OS/2 applications, a large .EXE file is appropriate as segments are demand-loaded. However, if a project is very large and sections of code are shared, it may be worth making these sections of code into DLLs. This reduces the amount of code on disk and speeds up the program make time.

DLLs may also be used to distribute updates. Rather than supply a new version of a program, separate DLLs may be updated as necessary.

Pitfalls of Dynamic Linking and their Solution

The following pitfalls should be noted:

- You must ensure that the interface between the executable file and the DLL is valid.
- As with all new ideas, there is a penalty to pay; new concepts need to be understood and appreciated before DLLs can be used effectively and efficiently.
- Typesafe linking to DLLs is not possible, as the new executable file format does not contain fields that can be

used to specify the type and number of parameters to a function call. Special care must be taken to use the correct header files in C/C++ programs, .DEF files for Modula-2 programs or .ITF files for Pascal programs.

- If the DLL standard library is used, the advantages of smart linking within the library can be reduced, although library segments will still only be loaded on demand. A custom library DLL may be created either by removing unwanted modules or by creating a new module definition file.

Instead of using the library DLLs, it is also possible to link one user DLL in a project with the multi-thread model library and export the required functions to other DLLs. These function names should be listed in the .EXP file. This is a special file used by the linker to establish the necessary connections between calls to dynamically-linked functions in the executable file and their object code in the DLL.

Understanding Dynamic Linking

The easiest way to appreciate the advantages of DLLs is by first examining traditional *static linking*.

Static Linking

Traditional operating systems expect the program file to contain all the instructions necessary to run that program. If libraries of procedures are used, they must be bound with the main program using a linker before the program can be run. When the libraries change, the program must be re-linked in order to generate a new run-time version. Each executable program thus carries with it a copy of some part of the library.

Every time you write a new utility program, even if it is only a few lines long, it uses facilities from the standard libraries. Each program thus contains copies of routines extracted from the library; this is clearly inefficient, with many copies of the same code cluttering up your disk. For example, ten disk utilities would contain ten copies of the disk access procedures and program startup code from the standard library.

Linking the Traditional Way

Linking joins together object modules from compiled programs and supplied object libraries, to make stand-alone executable files. The linker examines the object file produced by the compiler, and attempts to bind referenced symbols with symbols that are defined elsewhere.

The operation of a traditional linker can best be visualized using an example:

Assume that the standard MATH library contains a function procedure fact. This function calculates the factorial of a number. We can, therefore, write the Modula-2 program below to generate the factorial of 3. The fact procedure is defined in the MATH module:

```
MODULE FactTest;
  IMPORT MATH, IO;
BEGIN
  IO.WrStr("The factorial of 3 is");
  IO.WrInt(MATH.fact(3),3);
  IO.WrLn;
END FactTest.
```

In C the fact function might be in a MATH library and the equivalent would be:

```
void main( int argc, char *argv[] )
{
  extern int fact( int number );
  printf("Factorial %d is %d\n",3,fact(3));
}
```

This program could then be compiled in the usual way to generate an object file. This object file contains, amongst other things, an indication that the program needs to use a procedure called fact, which is stored in the MATH library (see figure .1).

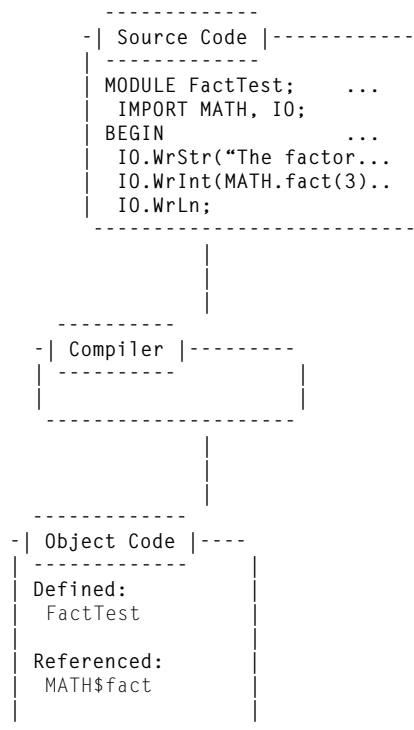


Figure .1 The compilation process, showing symbol references and object file definitions

The compiler identifies two types of symbols in a source program:

- *Defined Symbols*, which are the names of procedures which you have defined in your program.
- *Referenced Symbols*, which are the names of procedures which you have referenced in your program.

The object file contains explicit information regarding these two types of symbols. The linker uses this information to build the executable file. This is shown in Figure .2. In reality there would be far more references than are shown, but they have been omitted for the sake of clarity.

The object file created by the compiler must now be passed through the linker to allow the *referenced* symbols to be bound with the appropriate subroutines from the libraries. As a result, the linker is able to build an executable file.

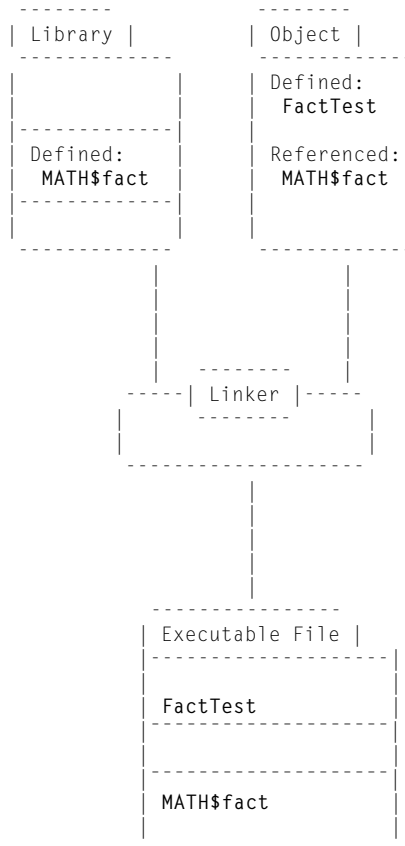


Figure .2 Binding referenced symbols to defined symbols

The executable file thus contains a copy of the compiled version of fact. If at a later time a bug were to be discovered in the library version of fact, you

would have to re-link your program to take this into account. In addition, you would have to re-link *every* program that uses fact, as each program contains a copy of this code.

The other drawback to traditional linking concerns hardware changes. If an application is written for a machine with a monochrome monitor and is subsequently updated for a color screen, you must provide one of the following:

- Drivers for a wide range of monitors,
- A totally new version of the program.

Electing to provide drivers means that new drivers must be written for each type of video screen on which the program is to be run. The executable program then has to contain (or have access to) a large number of different screen drivers, only one of which is ever used. You also have to adjust every application for the current hardware configuration.

Dynamic Linking

Dynamic linking addresses the problems associated with static linking in two ways:

- Late binding
- Shared code and data

Late binding delays the linking of a module's code until the program is started. This means that the very latest version of a procedure can be used and the version is optimized for your present hardware configuration.

Shared code and data allow the operating system to use a machine's available memory more efficiently. Only one copy of the procedure needs to be stored in memory, no matter how many programs are using it. With the advent of virtual memory, this technique can greatly reduce the operating system's requirement to swap areas of memory to disk. In the case of fact, the advantages are negligible, but for other procedures (such as graphics procedures) the gains can be very considerable.

A dynamic linker achieves this by storing a *reference* to the external procedure in the resulting executable code. Like a static linker, it verifies that the external procedure can be referenced. However, unlike a static linker, it does not store any code in the file for such a procedure, just a reference to its name. The burden of adding the code is transferred from the linker to the loader.

The basic MS-DOS loader is quite simple Æ it reads a file into memory and runs it. Using DLLs, the loader assumes a new set of responsibilities.

The loader must:

- Read the executable file and look for dynamic link references.
- Check to see if the dynamic link modules referenced in the file have already been loaded, since under OS/2 they can be shared with another application which may already be in memory.
- For any modules not yet loaded, search the available dynamic link libraries for the specified modules and load them into memory.
- Perform any necessary initialization for the modules that have been loaded.
- Run the program.

While DLLs offer many advantages, it would be extremely inefficient if all modules were loaded in this manner. There is still a need for some static linking when using DLLs. It is your decision whether a particular module is loaded dynamically or statically. You should treat each case independently.

As with any innovation, the use of DLLs imposes some constraints and rules on the modules that are placed in the Dynamic Link Libraries. You must carefully consider the requirements of a module being written.

Writing Dynamic Link Library Modules

There are several points to consider when writing dynamic link library modules, which are discussed below.

There are virtually no programming differences. However the linkage process itself, as explained in the preceding section, is slightly more complex and you must take certain steps to guarantee its secure operation.

In particular you should be aware that the linker will create not one but two files: the .DLL file itself which contains the object code, and an *Import Library* with a .LIB extension. This lists the functions which are to be found in the .DLL and is needed when the linker is dealing with a program that uses the .DLL. It supplies the linker with the information it needs to insert calls to .DLL modules into the final .EXE file.

Provided that this is correctly specified, the dynamic linker takes care of all the problems and ensures the correct behavior of the program.

The loader itself has to be able to access the .DLL at run-time, if necessary via the LIBPATH configuration variable.

If the loader cannot find the required DLLs when the program is loaded, an error is reported and the program aborted. As far as you are concerned, the

use of procedures from dynamic link modules is identical to using any other procedure. The procedure is called with the required parameters and returns a value.

The loader takes care of bringing the required code (and, possibly, data) into memory at load time. How, and when, this is done need not be your concern unless you want to create your own DLLs.

At a more advanced level, it is possible to use DLLs not only at load time, but also while the program is running. This feature allows the program to select the modules required in response to the specific demands of the operations being carried out.

The above summary of the loading of DLL-dependent programs is a simplification of the process under OS/2. Under OS/2, DLL modules are always loaded at run-time, but can be identified either:

- When the program is loaded into memory (*Load-time Dynamic Linking*), or:
- While the program is running, in response to the operational demands of the program (*Run-time Dynamic Linking*)

Creating Dynamic Link Libraries

As long as your source code obeys a few simple rules (discussed below), it is possible to use exactly the same code for both static and dynamic libraries. All the changes are handled through the project file and TopSpeed's project facility.

The changes to the project file can be summarized as follows:

- #system must specify os2 dll.
- #model must specify either dynalink or mthread.

When creating DLLs, you have the option of linking with the DLL versions of the standard libraries. This is the most simple and often most efficient approach. In this case the model dynalink should be used.

It also possible to make a DLL that links statically with standard libraries as long as functions that do not require initialization are used. See the "*TopSpeed Advanced Programmer's Guide*". In this case mthread model must be used.

- You must include the DLL start up module initdll with every DLL. When using #link this is done automatically. If a specialized startup is required, see the section below on using initdll.a.
- A module definition file (.EXP) file must be created.

For OS/2 all that is required is a list of the symbols to be exported.

You need only export those symbols you wish the DLL user to “see”. This allows you to create private procedures in physically separate source files. This avoids your DLL source becoming vast and monolithic. Such references are, in static linking terms, public symbols; in DLL terms they are totally hidden from you. Only you, as the DLL creator, are aware of their existence and use.

Executing *Make* with a project file thus constructed produces the functional DLL. The DLL and its interface actually consists of two files:

- A .LIB file that contains the information necessary for the static linker to generate the correct information in an .EXE file
- A .DLL file which actually contains the code for the DLL modules and is required at run-time

As explained below (see ‘*Running Programs that use DLLs*’), the DLL needs to be placed so that the run-time linker is able to find it at run-time.

Creating Module Definition Files

A file with the extension .EXP and the same name as the DLL must be created. All that is required for an OS/2 DLL is a list of the symbols to be exported and the entry point number:

```
EXPORTS
Symbol1      @1
Symbol2      @2
```

Entry point numbers must be listed in numeric order. A shorthand version is accepted:

```
EXPORTS
Symbol1      @?
Symbol2      @?
```

Each entry point will be automatically numbered correctly.

A *module definition file* may be created from an object file by using the utility TSMKEXP. For the complete module definition file syntax see the “*TopSpeed Advanced Programmer’s Guide*”.

Multi-thread DLLs

The TopSpeed DLL version of the standard libraries are inherently multi-threaded. Thus you do not need a separate library to make multi-thread DLLs.

Using #implib

The Project System can produce an import library automatically using the `#implib` directive. See the section on '*The #implib Command*' in Chapter : '*The Project System*' for more information.

While removing the need to create a module definition file may seem attractive, control over the DLL interface is lost using this method. As an export list may be created very easily using TSMKEXP, the use of module definition files should remain the method of choice.

Creating Programs that use DLLs

When switching from static to dynamic linking, your project file must specify the `dynalink` model, if the DLL versions of the standard libraries are to be used, or the `mtthread` model if the main program links statically with the standard libraries. The import library (.LIB) file associated with each DLL used must be added to the link list. (If using `#link`, the import library files for the standard TopSpeed libraries are added to the link list automatically.)

The project file produced using these rules generates a .EXE file that is able to utilize Dynamic Link Libraries.

Provided that you follow these guidelines, you need not make any source code changes.

Running Programs that use DLLs

When a DLL-based program is run under OS/2, the appropriate DLLs *must* be available to it. In order to locate DLLs, the configuration variable `LIBPATH` is used. This specifies a path (or paths) to be searched for the appropriate libraries. This is similar to the use of the `PATH` environment variable to locate executable programs.

If the required DLL cannot be found in any of the specified paths, a DLL run-time error is reported.

An Example

Since the major consideration for implementing DLLs under TopSpeed is the correct settings in the .PR file, the examples given here concentrate on the necessary project file settings. The language source code is largely irrelevant.

Creating the DLL

The source module contains the procedure fibo, which is to be made available to programs using the DLL. The DLL is to be called MATHDEMO.

The source file looks like this:

```

DEFINITION MODULE mathdemo;

    PROCEDURE fibo( x : CARDINAL ) : CARDINAL;

END mathdemo.

IMPLEMENTATION MODULE mathdemo;

PROCEDURE fibo( x : CARDINAL ) : CARDINAL;
    (* compute the fibonacci function *)

    VAR
        a,b,c: CARDINAL;

    BEGIN
        a := 0;
        b := 1;
        WHILE (x > 0) DO
            c := a + b;
            a := b;
            b := c;
            x := x - 1;
        END;
        RETURN a;
    END fibo;

END mathdemo.

```

The C equivalent would be:

```

/* MATHDEMO.c : Demonstration DLL */

int fibo( int n )
    /* Calculate nth Fibonacci no. */
{
    switch (n) {
        case 1 : return (2);
        case 2 : return (3);
        default : return (fibo(n-1) + fibo(n-1));
    }
}

```

The associated project file (MATHDEMO.PR) to create the DLL looks like this:

```

#system os2 dll
#model dynalink
#compile %main
- other #compiles if necessary
#link %prjname

```

The associated module definition file looks like this:

Initialization in a DLL

The initialization procedure for a DLL program is necessarily more complex than that used by a single .EXE file. This will affect users of object-oriented language features, particularly C++ users, for whom it is helpful to be aware of the procedure.

Before process startup the initialization code defined in `initdll.a` is called. No particular ordering is guaranteed under OS/2.

Low level, library and static C++ object initialization is carried out first at process startup. If the process comprises multiple DLLs all constructors are executed at this time. The order of initialization between modules is undefined.

On termination destructors are called in reverse order.

All Modula-2/Pascal module and static object initialization is then carried out. If the process comprises multiple DLLs the startup code of all modules is executed at this time.

Using INITDLL

All the normal initialization mechanisms of Modula-2, Pascal and C++ are performed automatically, but if some initialization specific to a DLL is required it may be called from the `initdll.a` file linked into the DLL.

You must use the supplied version of this file as a template, because the call to `InitLink` must be preserved. A call to user code may be added.

A non-zero result must be returned to indicate successful initialization. For example:

```

select
                                mov bx,          INITCODE
                                mov ax,          Initrec
                                extrn            INIT_DATA
                                call far         __InitLink
                                extrn            _mycode
                                call far         _mycode      (* returns !0 on success
                                *)
                                ret far         0

```

Linking the Initialization File:

If you are using a modified version of `initdll.a` with the `#link` directive in your project file no further action is necessary.

If you wish to use a custom project file using `#dolink`, your project file must contain the lines:

```
#compile myfile.a - user initialization file
#pragma linkfirst(myfile.obj)
```

Restrictions:

No library modules or objects will have been initialized when this code is executed.

Rules and Limitations for DLL Programs

The following restrictions apply regardless of the source language you are using:

- All pointers must be *far* pointers. Using *near* pointers for data pointers causes problems, since the automatic data segments are not combined by the dynamic link loader (unlike with a static linker). Setting the model to *extra large* ensures that the pointer type defaults to *far*. If you wish to use smaller, 16-bit, pointers, you will have to use *Segment Based Relative Short Pointers*. These are explained in detail in Appendix A: '*Memory models*'.
- All the exported procedures in a DLL must be accessed with *far* calls (and must, therefore, use *far* returns).
- Programs must use a *far* stack and, if present, the `ss_in_dgroup` pragma is ignored. This is in line with keeping all pointers as *far* pointers. The `stack_size` pragma, however, may be used normally to set the size of the stack.
- Data areas can be referenced and defined in both DLL modules and programs with the one condition that all pointer references are made using *far* pointers.

Library Usage and Restrictions

The following restrictions should be noted:

- *Near* heap functions are not available when using *dynalink* model.
- The Modula-2/Pascal `SetJump` and `Longjmp` procedures and `LongLabel` structure (In C, the `setjmp` and `longjmp` and the `jmpbuf` structure) can be used without restriction as long as the above rules are obeyed.
- `atexit` can be used to generate exit lists without any new restrictions.

Dynamic Link Loader Error Messages

For errors under OS/2 see Microsoft documentation.

Distributing DLLs

If you intend to distribute your DLLs to a third party, please read the License Statement which comes with the distribution disks.

CHAPTER 8

Multi-language Programming

This chapter provides the following information:

- Linking with standard libraries belonging to other languages.
- Creating your own inter-language interface files
- Type equivalents.
- Calling conventions.
- Naming conventions.
- Library considerations.
- Program startup and termination.

This chapter cannot be a tutorial for all TopSpeed languages, thus a knowledge of the languages concerned is assumed.

The interface to assembly language is described in the *“TopSpeed Advanced Programmer's Guide”*.

Standard Cross Definition Files

Cross definition files for all language library interface files are available to access the library functions of other languages.

These files may be imported or included into program source when it is written. Their inclusion will automatically cause the correct libraries to be linked when the program is made.

For example, calling the function `printf` from the TopSpeed C library from within a TopSpeed Modula-2 program module requires you to import the appropriate definition file `stdio.def`:

```
MODULE UsePrintf;  
  
IMPORT stdio;  
  
BEGIN  
    stdio.printf('Hello World');  
END UsePrintf.
```

`printf` can then be used in just the same way as if the program were a regular C file.

Creating your own Cross Definition Files

There are three main tasks involved in the creation an of an efficient inter-language interface:

- The translation of the appropriate declarations.
- The setting up of the necessary calling conventions.
- The setting up of the appropriate naming conventions.

The cross-library definition files supplied with your TopSpeed product are the best illustration of how the various inter-language interfaces should be declared.

Linking with your own Libraries

If the standard library cross definition files are used to access TopSpeed libraries from other languages, the correct library files will automatically be linked. However, if you make your own library requiring the standard library from another language, a project pragma should be inserted in to the interface files for your library. For example:

```
#pragma project("#set tsm2=on")
(*# project("#set tsc=on") *)
```

The above pragmas can be used to add the appropriate library to the link list, depending on the macro set:

tsc=on	Link C libraries
tscpp=on	Link C++ libraries
tsm2=on	Link Modula-2 libraries
tspas=on	Link Pascal libraries

It may also be necessary to explicitly link the C graphics or text window libraries if they are used:

cwindow=jpi	Use C text windows
cgraph=jpi	Use C graphics

Note. These macros are processed by the #link command to determine what libraries to include. If your project file uses #dolink, you will have to determine the appropriate libraries and add them to the link list yourself.

Type Equivalents

The following table lists the type equivalences between the TopSpeed language products:

C & C++	Modula-2	Pascal
signed char	SHORTINT	int8
unsigned char	SHORTCARD	byte
char	CHAR	char
unsigned char	BOOLEAN	boolean
unsigned int	CARDINAL	word
int	INTEGER	int16
unsigned short	CARDINAL	word
short	INTEGER	int16
unsigned long	LONGCARD	integer
long	LONGINT	integer
float	REAL	shortreal
double	LONGREAL	real
long double	TEMPREAL	longreal
void near *	NearADDRESS	nearaddress
void far *	FarADDRESS	faraddress
void *	ADDRESS	address
char * (string)	ARRAY OF CHAR	string

Strings

Modula-2 to C

In C, all strings are considered to be zero terminated. In Modula-2, however, strings will be zero terminated unless their length is equal to the size of the array. It is essential that strings passed from Modula-2 to C are properly zero terminated. The function `Str.StrToC`, supplied as part of your TopSpeed product will create a Modula-2 string compatible with the C language.

The C language passes strings as a simple pointer to char, with no array size information. Therefore the call pragma must be used to disable the passing of the array size when using Modula-2 strings in C programs:

```
(*# call(o_a_size=>off, o_a_copy=>off) *)
```

C to Modula-2

Due to the reasons discussed in the previous paragraph, no special translation is required when passing strings from C to Modula-2. However, the array size must be passed explicitly:

```
unsigned Str$Length(unsigned size, char *s);
```

Pascal to C and Modula-2

In Pascal the dynamic string type is an array of type char. Element 0 is recognized as the dynamic length and element 1 as the first element of the string. The TopSpeed procedure `StrToZ` will convert a Pascal string to a type suitable for passing to either C or Modula-2.

When passing a Pascal translated string to Modula-2, a typeless var parameter must be employed with the `call(t_l_size=>on)` pragma:

```
(*# call(t_l_size=>on) *)
function Length(var s): word;
```

When passing a translated Pascal string to C, a typeless var parameter is used with the pragma `call(t_l_size=>off)`:

```
(*# call(t_l_size=>off) *)
function Length(var s): word;
```

C to Pascal

When calling a Pascal function and passing a string, the size of the array must be passed as a byte before the address of the array, and byte 0 must contain the dynamic length. The function `StrToPas`, declared in `mlang.h`, will achieve this.

```
void Pasfunc(unsigned char size, char *s);
```

Modula-2 to Pascal

The Modula-2 string may be translated to Pascal format by employing the procedure Str.StrToPas.

Although the calling conventions used in the two languages are almost the same (a size parameter followed by the address of the array) a type inconsistency error will occur due to the type of the size parameters differing. As a result of this, successful passing requires that each parameter must be passed explicitly and the pragma call(o_a_size=>off) used:

```
(*# call(o_a_size=>off) *)
PROCEDURE PasFunc(size: byte;
                  s: ARRAY OF CHAR);
```

Enumeration Types

Enumeration sizes in C/C++ and Modula-2/Pascal are not compatible by default, as C and C++ use a 16-bit type, while Modula-2 and Pascal use an 8-bit type.

The call(var_enum_size=>on) pragma must be used in Modula-2 and Pascal to achieve compatibility.

Calling Conventions

As well as setting up the string calling conventions mentioned above, the overall calling convention must be specified whenever multi-language programming is employed.

While the basic JPI calling convention is consistent between all languages, use of the correct pragma declarations will ensure that an interface file is valid using the stack frame convention as well.

Modula-2 to C

The following pragmas define the C or C++ calling convention for a Modula-2 program:

```
(*# call(o_a_size=>off,o_a_copy=>off,
         c_conv=>on,          result_optional=>on) *)
(*# module(implementation=>off,
         init_code=>off) *)
```

C to Modula-2

The following pragma defines the Modula-2 calling convention for any TopSpeed C or C++ program:

```
#pragma call(c_conv=>off)
```

Pascal to C

The following pragma defines the C or C++ calling convention for any TopSpeed Pascal program:

```
(*# call(t_l_size=>off,t_l_copy=>off
      c_conv=>on,      result_optional=>on) *)
(*# module(implementation=>off
      init_code=>off) *)
```

C to Pascal

The following pragma defines the Pascal calling convention for any TopSpeed C or C++ program:

```
#pragma call(c_conv=>off)
```

Pascal to Modula-2

The following pragma defines the Modula-2 calling convention for any TopSpeed Pascal program:

```
(*# call(t_l_size=>on) *)
(*# module(implementation=>off) *)
```

Modula-2 to Pascal

The following pragma defines the Pascal calling convention for any TopSpeed Modula-2 program:

```
(*# call(o_a_copy=>off, o_a_size=>off) *)
(*# module(implementation=>off) *)
```

Naming Conventions

To achieve the correct linkage names, the name pragma must also be used.

Modula-2 and Pascal to C

The following pragma defines the C and C++ naming convention for any programs written using TopSpeed Modula-2 or TopSpeed Pascal:

```
(*# name(prefix=>c) *)
```

C to Modula-2 and Pascal

The following pragma defines the Modula-2 and Pascal naming convention for any program written using TopSpeed C:

```
#pragma name(prefix=>"" )
unsigned Str$Length(unsigned size, char *);
unsigned MyModule@CardinalVar;
```

Names in TopSpeed Modula-2 and Pascal are overloaded between modules, so it is often best to include the module prefix in the identifier.

- Procedure names use \$ as a separator.
- Variable names use @ as a separator.

If it can be guaranteed that no name clashes will occur, the module prefix can be set explicitly in the pragma.

```
#pragma name(prefix=>"MyModule@")
unsigned CardinalVar;
```

C++ to Modula-2, Pascal and C

When using TopSpeed C++ a linkage specifier should be used:

```
extern "Modula2"
extern "Pascal"
extern "C"
```

C++ linkage specifiers for Modula-2 and Pascal may also contain a module name. For example:

```
extern "Modula2.MyModule"
extern "Pascal.MyUnit"
```

Modula-2 to Pascal, Pascal to Modula-2

No action is necessary. The naming conventions are identical apart from case restrictions.

Library Considerations

There are a number of important points which must be considered when preparing multi-language programs.

I/O

File handles may be shared between C and Modula-2, as may stream buffer descriptors.

File handles of unbuffered files may be freely passed between Modula-2 and C. However if the Modula-2 File variable refers to a buffered file the stream pointer (FILE*) variable must be used. See FIO.GetStreamPointer and FIO.AppendStream in the "*TopSpeed Modula-2 Reference*".

Care must be taken that streams or files have the same access and buffering modes.

The Pascal I/O paradigm is different to that of C and Modula-2 and care must be taken if predefined I/O streams are shared.

Memory Allocation

All languages in a multi-language program share the same far and near heaps. Pointers may be freely exchanged.

Warning

In Modula-2 the heap overhead is increased from 0 to 2 bytes per allocation.

Window Modules

The C, C++ and Modula-2 window modules are compatible and handles may be interchanged freely.

The Pascal TurboCrt and C/C++ clipping window modules are compatible.

The Pascal PasWin module is not compatible with either the JPI or clipping window modules.

Process Modules

The C, C++, Modula-2 and Pascal process multi-thread modules are compatible.

Program Environment Variables

Any changes that are made to the environment by the C function putenv will NOT be reflected in the values returned by the Modula-2 environment functions.

Overall Order of Library Initialization

The initialization of library low level modules, static objects and Modula-2 and Pascal modules occurs before the execution of the program starting point Æ the function main or the main module:

1. **Low-level system startup.**
2. **Library low level initialization.**
3. **C++ library static objects.**
4. **User C++ static objects.**
5. **Modula-2 and Pascal module and static object initialization code.**

Program Termination

On program termination the following procedures are executed:

1. The Modula-2/Pascal terminate chain is executed.

2. Any procedures on the C atexit/onexit stack are executed on a last in first called basis.
3. Any C++ static destructors are called in the reverse order to that in which the constructors were called.
4. Low level library cleanup is executed. Files are flushed and closed, then temporary files are deleted. Interrupt vectors are restored.

If a new process is executed (using the C exec??? family of functions) interrupt vectors are restored and open streams are flushed. No user terminate functions are called and static destructors are not called.

Termination due to Fatal Error

The normal termination procedure is followed and then the ERRORINF.\$\$\$ file is created. If another fatal error occurs during processing of termination code the process terminates immediately.

Program Termination Under OS/2

The above procedures are followed except in the case of an exception such as a segment over-run:

- Only user specified termination procedures are executed (those installed by Terminate or atexit).
- Code should be limited since a recursive error will prevent OS/2 from killing the process.
- By default buffered streams will not be flushed and static C++ destructors will not be called.

Modula-2 and Pascal Initialization from C

Any modules used from within a C compilation unit will NOT be initialized unless imported by a Modula-2 main module, or one of its imports. If your main module is in C then the function InitModules, declared in mlang.h, must be called from the function main() before any Modula-2 or Pascal functions are called.

```
void InitModules(char InitData1[], ...);
```

The function takes a list of one or more module initialization identifier arrays, terminated by a NULL pointer. For example:

```
#include <mlang.h>

main() {

    InitModules(FIO$, Lib$, MyModule$, NULL);
    ...
    return 0;
}
```

The identifiers for the Pascal and Modula-2 libraries are declared in mlang.h.

Similar declarations should be made for any of your own modules that have initialization code, i.e. module(init_code=>on):

```
#pragma save
#pragma name(prefix=>"" )
/* Must not have _ prefix */
extern char MyModule$[];
#pragma restore
```

Multi-language Object-Oriented Programming

Modula-2 and Pascal

The only fundamental difference between Modula-2 and Pascal objects is that in Modula-2 methods are static by default, and in Pascal methods are dynamic by default.

Modula-2/Pascal and C++

The following features are only available in C++:

- Virtual base classes.
- Overloaded methods.
- Overloaded operators.
- Pointers to members.

The following features are only available in Pascal/Modula-2:

- Checked guard operator.
- The IS class test.
- A parameterless constructor is always present.
- The 'size' of an object (instantiated class), gives you the dynamic size of the object.

As a consequence of the above it is not possible to interface to a C++ class from Modula-2/Pascal, if that class contains virtual base classes or pointers to members. Also it is not possible to use the guard operator or the IS operator on a class which is implemented in C++, or on a class which is derived from a class implemented in C++. C++ and Pascal/Modula-2 also, by

default, use different representations for method tables. These formats are documented in Appendix B: ‘*Class Object Formats*’.

For the method tables to be compatible, the following pragmas have to be used:

```
(*#data(class_hierarchy=>off,
      cpp_compatible_class=>on)*)
```

in Pascal or Modula-2.

```
#pragma data(compatible_class=>on)
```

in C++.

Furthermore, the order in which methods are defined must be the same in all the languages, and the order in which base classes are inherited in a derived class must be the same. A C++ class must contain a parameterless constructor, and an interface in C++ to a class implemented in Pascal/Modula-2 must contain a parameterless constructor.

Naming of Methods

The naming of virtual methods is not important, since it is the location in the method table which determines which method is called.

However, non-virtual methods represent a problem. As a consequence of overloaded methods in C++, the C++ compiler uses encoded names, as documented in the “*TopSpeed C++ Language Reference Manual*”. If a class does not have any overload methods, you can use the C++ linkage specification “pascal” or “modula2”, which means that the C++ compiler will not produce encoded names:

```
extern “pascal.module_name” {class definition}
```

However, if the class does contain overloaded methods, you will have to use the encoded names in the Pascal or Modula-2 definition. An encoded name consists of a prefix, the name, and a postfix (the parameter profile). If the name(prefix=>c) pragma is used in Pascal/Modula-2, the Pascal/Modula-2 compilers will generate the prefix automatically. When you create an interface for Modula-2/Pascal you must include the postfix in the name yourself.

Note: The linker will produce both encoded and the equivalent C++ function names side-by-side in the MAP file, which can be of assistance in this process.

Access to C++ operator functions can be gained by using the C++ encoded names for the operators (See the “*TopSpeed C++ Language Reference Manual*”). The same is true for constructors with parameters.

Programming Examples

The example programs MIX_PAS and MIX_MOD demonstrate the C++ to Pascal and Modula-2 interfaces respectively.

The example program MIX_M2P demonstrates the Pascal to Modula-2 interface.

CHAPTER 9

Interfacing to Third-Party Code

TopSpeed supports, and is supported by, many third party products. The list is constantly growing, and so is not documented here. The file LIBS.DOC, found in the TS\DOC directory contains the latest information on supported third party products.

Interfacing to Existing Code

If you have some assembly language modules, or code generated by another compiler, it is necessary to declare the correct calling convention.

The Standard C/C++ Calling Convention

The correct stack frame calling convention must be declared for the function prototypes concerned:

```
#pragma save
#pragma call(reg_param=>(),
            c_conv=>on,
            reg_saved=>(di,si,ds,st1,st2)
int standard_call_func(int a, long b);
#pragma restore
```

The above function is declared as using the standard C stack calling convention, and preserving the standard register set.

The file pragma.h defines many useful pragma settings, and may be used to declare both jpi and standard calling conventions:

```
#pragma save
#pragma _CDECL_CALL
int standard_call_func(int a, long b);
#pragma restore
```

The Standard Modula-2/Pascal Calling Convention

The correct stack frame calling convention must be declared for the declarations concerned:

```
(*# save,
   call(reg_param=>(),
        c_conv=>off,
        reg_saved=>(di,si,ds,st1,st2)
*)
PROCEDURE StandardProc(a: INTEGER);
(*# restore *)
```

The above function is declared as using the standard stack calling convention, and preserving the standard register set.

Using Stack Frame Libraries

In some circumstances, merely declaring a function or procedure that uses the standard calling convention will not be enough to get your program working. In the following circumstances it will be necessary to use a stack frame library:

- When a function or procedure in a standard calling convention module calls back into the standard library.
- When the function or procedure called requires more than one level of floating point stack.

A large model stack frame library is supplied as standard, and other models may be built by TechKit[®] and SourceKit users.

Assembly Language Interface

The complete assembly language interface to TopSpeed languages is documented in the “*TopSpeed Advanced Programmer’s Guide*”.

CHAPTER 10

Presentation Manager Programming

Building programs to run under OS/2's Presentation Manager is automatic using the TopSpeed Project System. However, documenting the Presentation Manager (PM) API is beyond the scope of this manual. Please refer to the appropriate OS/2 documentation.

Program Source

The pragma `call(same_ds)` must be set correctly on all call-back procedures and functions, to ensure that the correct data segment value is used during execution.

```
call(same_ds=>off)
```

The `data(heap_size)` pragma should also be used to specify the program's near heap requirement.

```
pragma data(heap_size=>4000)
```

C and C++ Languages

The PM interface is declared in `os2pm.h` and its sub-include files. It may be necessary to enable some declarations by defining macros. These are documented at the head of the include file concerned.

Modula-2 and Pascal Languages

The PM interface is declared in various modules, and is documented in the relevant "*Library Reference Manual*".

Making PM Programs

A project file for a PM program only needs one addition to the standard OS/2 project. Three Project System macros are available to control PM compatibility:

<code>pm_api</code>	makes a program of the OS/2 application type <code>WINDOWAPI</code> .
<code>pm_compat</code>	makes a program of the OS/2 application type <code>WINDOWCOMPAT</code> .
<code>pm_noncompat</code>	makes a program of the OS/2 application type <code>WINDOWNOTCOMPAT</code> .

A typical project file would therefore look like this:

```
#system os2 exe
#model small
#set pm_api="on"

#compile %main
#link %prjname.exe
```

A PM program may also be made without using the automatic #link command:

- The PM API import library is pmjpi.lib. This must be added to the link list by using #pragma link(pmjpi.lib). See Chapter : *'The Project System'*.
- The startup file initpm.obj must be used in place of initexe.obj. This is achieved by using #pragma linkfirst(initpm).
- The correct setting of the link_option(pm) and link_option(non_pm) pragmas must be selected. See the section on *'Link_option pragmas'* in Chapter : *'Pragmas'*.

Using a Resource Compiler

The standard OS/2 resource compiler does not use the TopSpeed redirection system. Therefore, the INCLUDE environment variable should be set correctly.

CHAPTER 11

Using CodeView and Compatible Debuggers

For most purposes, the TopSpeed debugger VID will be found preferable to debuggers produced by third party suppliers, for debugging TopSpeed code. However, for some applications, in particular when debugging under Windows, the use of the Microsoft CodeView debugger may be necessary. Also, many hardware-supported debuggers can make use of debug information in CodeView format. TopSpeed therefore includes a utility program, VID2CV, to convert from VID debug information into CodeView format.

Using VID2CV

The use of this conversion program is extremely straightforward. From the DOS or OS/2 command line, simply type:

```
VID2CV progname
```

where progname names the executable file to be converted for use with CodeView. VID2CV will report on progress as it reads the executable file, any corresponding .DBD files it can find, and the .MAP file if available. The executable file is modified so that it can be used with CodeView. The VID information in the file is left intact, so that VID can still be used if required.

If you regularly use CodeView on a particular file, you may want to add VID2CV to the project file, for example:

```
#link myfile  
#if (%make=on) #and #not (%mode=compile) #then  
    #run "vid2cv myfile"  
#endif
```

If you wish to use the MicroSoft resource compiler with an executable file that contains debugging information, you must use VID2CV before executing the resource compiler, otherwise the VID information is stripped.

Limitations

Because CodeView was never designed to support Pascal or Modula-2, certain structured types (notably SET types) that are correctly displayed by VID will be translated into ARRAY OF BYTE in the CodeView information. This allows the information to be accessed in CodeView, but the interpretation of this information must be done by you. VAR parameters are displayed as pointer types by CodeView, and must be dereferenced in order to obtain the variable's contents.

CHAPTER 12

Miscellaneous Programming Considerations

Extending File Handle Limits

Select the appropriate corefile.XXX interface file for your language and edit as instructed in that file.

The file must then be included in your project file:

```
#compile corefile
```

This generates linker warnings, since two definitions of the variables in corefile exist. These may safely be ignored, because the linker will use those defined in corefile.

OS/2 Multi-thread Programming

Under OS/2 three layers of multi-thread programming exist:

- The top layer comprises the JPI process module. All library modules and floating point are supported, plus portability is possible to DOS.
- The middle layer involves using `_beginthread` and `_endthread`. (In Modula-2 and Pascal these are available in CoreProc). All library modules and floating point are supported, but portability to DOS is lost. Using this interface it is possible to determine the thread number of a child thread.
- The lowest layer is the OS/2 API. Using `DOSCREATETHREAD` does not initialize the library, floating point emulator and stack checking mechanism. Only those library functions that do not require locking or floating point may be used (for example, string functions).

APPENDIX A

Memory Models

In any computer system above a minimum level of complexity, applications do not use the addresses they are given to access memory directly. Instead, they are first translated (by hardware) to address a different physical memory location. This is at the heart of a modern operating system's ability to handle several users, to swap tasks in and out of fast memory, and to protect user programs against each other.

Thus, there is always a distinction to be made between the user address space, sometimes called the *virtual address space*, and the system address space, sometimes called *absolute address space*. Generally, the more complex the system, the more important this distinction becomes.

TopSpeed compilers run on the industry standard 80x86 family of microprocessors, which have a unique segmented architecture with a wide range of address modes. The compiler offers a full range of features to take advantage of the choice this offers and to optimize your program's use of memory and time, even on the most advanced operating systems now in use on the 80x86 range.

The 80x86 Architecture — a Design Compromise

The 80x86 architecture arises from a conflict between two design goals. Early microprocessors catered for compact programs which used memory efficiently. But as programs got bigger and memory got cheaper, it became both desirable and feasible to provide a large address space.

The 80x86 family offers both 16- and 32-bit memory addresses, and nine address registers which can be used in 36 different combinations. In addition the 80286, 80386 and 80486 processors offer two address modes, *real* and *protected*. Data and program can therefore be organized and referenced using a variety of different addressing schemes or memory models, each with its own advantages and drawbacks.

As a program writer you face a choice: if you want small, fast programs you must restrict what they can do and the size of the objects they can handle. However, if you choose to remove these restrictions, you will pay a price in efficiency. When choosing a memory model, you must try to strike the appropriate balance between these objectives.

Modern high-level languages, which are designed to be machine-independent, do not possess inbuilt program constructs to deal with this embarrassing wealth of address modes. For languages and programs which

do not use pointers, this is not a major problem. However, Pascal, C, Modula-2 and C++ offer extensive, and at times exotic pointer facilities, which are almost indispensable to modern object-oriented approaches.

Assembly Language Interface

A more advanced discussion of memory models for assembly language programmers can be found in the *“TopSpeed Advanced Programmer’s Guide”*.

TopSpeed Language Extensions for Memory Models

TopSpeed provides a comprehensive and uniform set of language extensions to deal with the 80x86 address structure so that programmers can profit from the choices offered by industry standard machines. This makes it the ideal tool for object-oriented programming and multi-tasking systems.

The Standard Memory Models

You do not have to familiarize yourself with different memory models to write working programs. TopSpeed has default standard models, which cater for the most common situations. For the great majority of applications, all you have to do is choose the correct standard model and the TopSpeed system will do the rest for you.

The 8086 Architecture

Address architecture is easier to understand by studying how it evolved from the original 8088 and 8086 system. This should provide you with an insight into the reasons why different memory models are required, and will help you to achieve the most efficient results from TopSpeed compilers.

Segments

The Intel 8088 and 8086 processors, which were used in the early, ‘standard’ PCs, in PC XTs and are still used in many laptops, have 20-bit address buses. This means that they can address up to one megabyte of memory. However, the processor registers only have 16 bits, so an absolute address requires two registers. Intel solved this problem by dividing the memory into 64K byte chunks called *segments*. These can begin anywhere in memory and be of any size under 64K bytes; they can even overlap.

Models which use multiple segments for code or data can obviously be larger. However, they incur an overhead of code size and execution time, since segment registers have to be managed by the program.

Near and Far Pointers

A program that only uses one segment for its data may access that data using a 16-bit pointer (a *near* pointer). Similarly a program that places all its code in one segment may call that code using a 16-bit pointer.

Conversely, a program that uses many segments for its data must access that data using a 32-bit pointer (a *far* pointer). A program that places its code in many segments must also call that code using a 32-bit pointer.

The management of pointer sizes is handled automatically by the compiler, according to memory model. The following section describes the standard memory models available when programming using TopSpeed.

The Standard Memory Models

TopSpeed compilers provide six standard memory models. The *small model* is the default memory model. Use the Project Memory models menu to change memory models. The following table summarizes the standard memory models:

Model	Small	Compact	Medium	Large	XLarge
Code size	64K	64K	1Mb*	1Mb*	1Mb*
Data size	64K	1Mb*	64K	1Mb*	1Mb*
Code pointers	near	near	far	far	far
Data pointers	near	far	near	far	far

* 16 Mb under OS/2.

- Each memory model has its own library, which is automatically linked in when using the project facility.
- The *multi-thread model* is the extra large model with support for the re-entrant library.
- The *overlay model* is the *multi-thread* model plus segment based overlaying.
- The *dynalink model* is the *multi-thread* model with dynamic linking. Under DOS it also includes segment-based overlays. (Dynamic linking under DOS is available with the TopSpeed TechKit[®].)

These models are described in detail below. On the left side of each diagram you will find a diagram of your program's address space, subdivided into segments. Each fresh segment begins with the segment name and class, in the format:

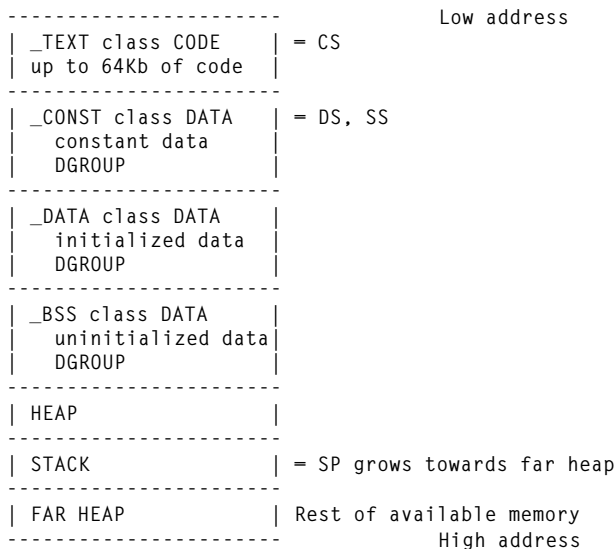
```
<Segment Name> class <class-name>
```

To the right of the diagram you will find an explanation of the segment and pointer registers used to access the segment.

Small Memory Model

The *small* memory model is the most efficient, and is the default. It contains one code segment of up to 64Kb and one data segment of up to 64Kb. The stack, global data and heap all use the default data segment. Code and data pointers are *near* pointers and all addresses are therefore 16-bit. Unoccupied address space outside the CODE and DATA segments is organized as a heap and can only be reached by creating explicit far pointers to it .

The following diagram illustrates the small memory model:



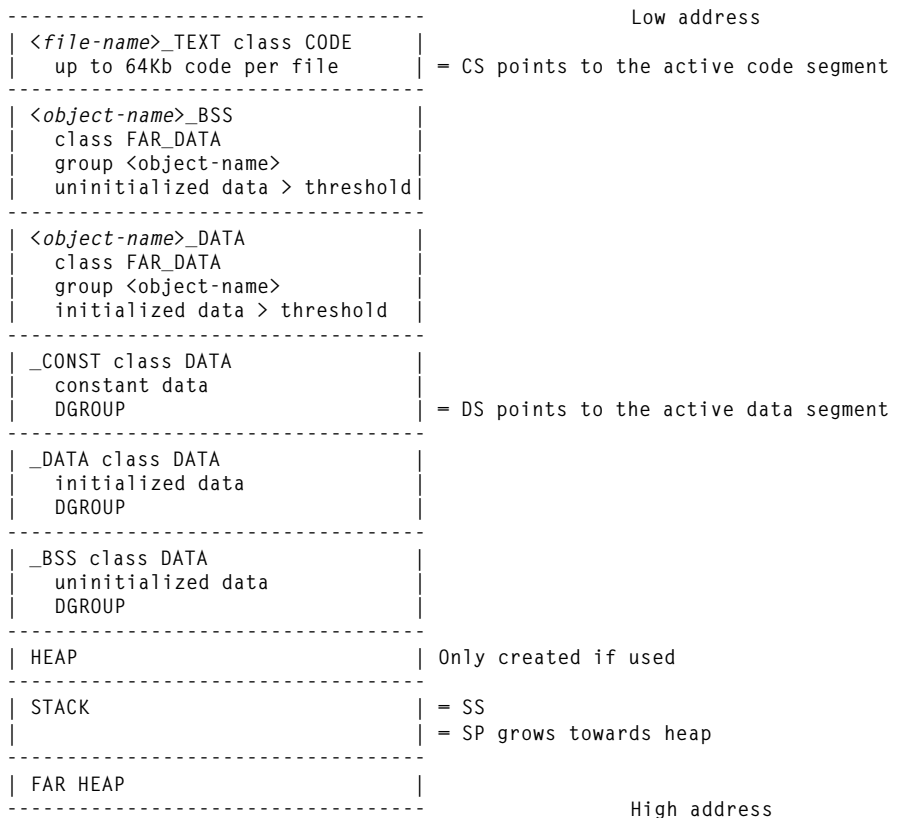
The CS register addresses the _TEXT segment. DGROUP, HEAP and STACK are all addressed through DS and SS. They may be up to 64Kb in total.

Large Memory Model

The *large* memory model can have multiple code segments and multiple data segments. All pointers are *far* pointers. There is one default data segment for all objects except for those larger than the *threshold*. The threshold is 32Kb by default and data objects greater than this have their own segments. Data objects smaller than the threshold go into the default data segment to aid performance. The following must be observed:

- No single data object may exceed 64Kb.
- No single source file should produce more than 64Kb of code.
- The sum of all data objects below the threshold must fit into 64Kb.

The following diagram illustrates the large memory model:



Each source file produces a code segment named `<file-name>_TEXT`, where *file-name* is the name of the object module. Data objects less than the threshold go into the default data segment DGROUP. Data objects greater than the threshold go into their own segments named `<object-name>_BSS` or `<object-name>_DATA`, where *object-name* is the name of the data object.

_BSS contains the uninitialized data objects and _DATA the initialized data objects. The DGROUP and HEAP must not exceed 64Kb in total. The near HEAP is only created when necessary.

Medium Memory Model

The *medium* memory model is like a large model for code with a small model for data. Like the small memory model, it is limited to 64Kb of data, but multiple code segments are allowed. It is used for large programs that do not have a large data space. Data pointers are *near* pointers, but code pointers are *far* pointers. As with the small model, DGROUP, the HEAP and the STACK are all addressed via DS or SS, and must fit into 64K

<div>-----</div> <div><file-name>_TEXT class CODE</div> <div>up to 64Kb code per file</div> <div>-----</div>	<div>Low address</div> <div>= CS points at active code segment.</div>
<div>-----</div> <div>_CONST class DATA</div> <div>constant data</div> <div>DGROUP</div> <div>-----</div>	<div>= DS,SS</div>
<div>-----</div> <div>_DATA class DATA</div> <div>initialized data</div> <div>DGROUP</div> <div>-----</div>	
<div>-----</div> <div>_BSS class DATA</div> <div>uninitialized data</div> <div>DGROUP</div> <div>-----</div>	
<div>-----</div> <div>HEAP</div> <div>-----</div>	
<div>-----</div> <div>STACK</div> <div>-----</div>	<div>= SP, grows towards HEAP</div>
<div>-----</div> <div>FAR HEAP</div> <div>-----</div>	<div>Up to the rest of available memory</div> <div>High address</div>

Compact Memory Model

The *compact* memory model is like a small model for code with a large model for data. It is limited to 64Kb of code but may have multiple data segments. It is used for smaller programs that address a lot of data. There is one default data segment for all data objects, but, like the large memory model, large objects can be placed in separate segments. Code pointers are *near* but data pointers are *far*.

-----		Low address
	_TEXT class CODE up to 64Kb code	= CS points at active code segment.

	<object-name>_BSS class FAR_DATA group <object-name> uninitialized data > threshold	

	<object-name>_DATA class FAR_DATA group <object-name> initialized data > threshold	

	_CONST class DATA constant data DGROUP	= DS

	_DATA class DATA initialized data DGROUP	

	_BSS class DATA uninitialized data DGROUP	

	HEAP	Only included if needed

	STACK	= SS = SP grows towards HEAP.

	FAR HEAP	Up to the rest of available memory High address

Extra-large Memory Model

The *extra large* memory model is an extension of the large model, in which each file has a data segment of its own.

-----		Low address
<file-name>_TEXT class CODE up to 64Kb code per file		= CS points at active code segment

<file-name>_BSS class FAR_DATA group <file-name>		= DS points at active data segment

<file-name>_DATA class FAR_DATA group <file-name>		

<object-name>_BSS class FAR_DATA group <object-name> uninitialized data > threshold		

<object-name>_DATA class FAR_DATA group <object-name> initialized data > threshold		

_CONST class DATA library constant data DGROUP		

_DATA class DATA library initialized data DGROUP		

_BSS class DATA library uninitialized data DGROUP		

HEAP		Only included if required by program

STACK		= SS = SP grows towards HEAP

FAR HEAP		Up to the rest of available memory High address

The *extra large* memory model is usually used when the *large* model is not adequate, because the total amount of data and constant items less than the threshold exceeds 64Kb.

Data objects smaller than the threshold go into the file's segment. Data objects greater than the threshold go into their own segments. The DS register is always loaded on entry to a function.

The multi-thread, overlay and dynalink memory models are all based upon the extra large memory model, and use the same conventions for code and data layout.

Multi-thread Memory Model

The *multi-thread* memory model has the same memory layout as the *extra large* model, but has its own re-entrant library. You should use the multi-thread library when writing multi-threading programs with the JPI time-sliced process scheduler or in OS/2.

Overlay Memory Model

The *overlay* memory model must be selected to build programs that use the TopSpeed Overlay Management System. Like the *dynalink* and *multi-thread* memory models, it uses the same conventions as the *extra large* memory model for segment layout, naming and pointers, but invokes additional pragmas and linker options to construct an overlay program.

The *overlay* memory model is a superset of the *multi-thread* memory model. This memory model is only available for DOS programs and DLLs - under OS/2 it is not required.

Dynalink Memory Model

The *dynalink* memory model is specified in order to use the dynamically-linked versions of the standard TopSpeed libraries. For OS/2 programs, the *dynalink* model is equivalent to the *multi-thread* model in all other respects, while for DOS programs it is equivalent to the *overlay* model.

The dynalink memory model may be specified when creating a DOS or OS/2 DLL, if it is required that the DLL should not contain code from the standard TopSpeed libraries. In this case, the DLL versions of these libraries will be used. A program using a DLL created this way would normally also be made using dynalink model.

Alternatively, a DLL may be constructed which does not make use of the TopSpeed libraries in their DLL form. In this case, the DLL should be built using the multi-thread (for OS/2) or overlay (for DOS) memory model. A program that used such a DLL would normally be made using the same memory model.

Strictly-speaking, the dynalink memory model is not a memory model, since it uses exactly the same segment organization and address system as the extra large memory model. However to all intents and purposes it appears to the programmer as a distinct memory model, since what it actually does is select a consistent set of pragmas which ensures that the segment organization, calling convention and addressing system work together correctly for interfacing to dynamically-loaded segments.

Selecting Memory Models

The standard and best way to select a memory model, is from the TopSpeed Environment using the Project Memory model menu option. This will automatically insert the correct pragma into your project.

Memory models may also be set from the compiler command line, or by means of a pragma in your program. In this case you must ensure that the model selected is consistent between the different modules used.

Selecting a Memory Model from the Command Line

You can select a memory model from the command line using the /mX switch, where X is:

s	Small model
m	Medium model
c	Compact model
l	Large model
x	eXtra large model
t	multi-Thread model
o	Overlay model
d	Dynalink model

If your choice of memory model is incompatible with the size and number of your data and code objects (for example, if you use a small model but supply data objects totalling more than 64K), the linker will signal an error.

Changing the Global Threshold

The compact, large, extra large and multi-thread memory models place objects larger than a threshold in a separate segment. The default value of 32Kb will not suit you if, for example, your program has three data objects of 30Kb. Indeed, these data objects will attempt to group into a single segment, which would then exceed 64Kb, which is not allowed. To overcome this, you need to alter the threshold size to below 30Kb. This can be done by inserting the following statement in your project file, after the #model command:

```
#pragma data(threshold=16000)
```

Near and Far Pointer Declarations

The syntax for these is slightly different for C/C++ and for Pascal/Modula-2. C and C++ provide a richer variety of data objects, and their pointer facilities are more extensive. Modula-2 and Pascal, on the other hand, impose stricter

controls on its data objects. However the underlying concept and implementation are identical.

In all languages the crucial construct is a modification to the pointer type so that it can be qualified as *near* or *far*. In C and C++ you are provided with *near* and *far* keywords, which qualify other declarators. These keywords always modify the object to the immediate right. For example:

```
char far *ptr;
        /* makes ptr a far pointer */
int (near *nFuncPtr) (int);
        /* makes nFuncPtr a near                function pointer */
```

TopSpeed Modula-2 and Pascal provide three generic pointer types. In Modula-2, these are defined in the SYSTEM module. In Pascal they are part of the language. Consider the following declarations:

```
VAR
  x : ADDRESS;
  y : NearADDRESS;
  z : FarADDRESS;
```

These declare three pointers, x, y, and z. y will be a 16-bit *near* pointer, whatever memory model the program is compiled with. z will always be a 32-bit *far* pointer of the form segment:offset. x will be NearADDRESS or a FarADDRESS depending on the memory model (or pragmas) used when the program is compiled.

Near and Far Arrays

In C/C++ arrays are treated as pointers. You can therefore declare a far array, which may affect how data is allocated. In all memory models where data objects are placed in the default data segment, the keyword *far* will place the object in its own separate data segment. For example:

```
static int far FarArray[100];
```

In all memory models where data objects are placed in multiple data segments, the keyword *near* will force the object into the default data segment. For example:

```
static int near NearArray[17000];
```

You can address an object in the default data segment with a *near* pointer. For example:

```
int near *nptr;  nptr = NearArray;
```

You can address an object in any data segment with a *far* pointer. For example:

```
int far *fptr;  fptr = FarArray;
```

Modula-2 and Pascal possess no equivalent construct, although the same effect may be achieved by using pragmas. See Chapter : '*Pragmas*' for further details.

Functions

C and C++ also offer an extra degree of control over function referencing.

You can call a function in the same segment as the calling function with a near call, because there is no need to reload the segment register. If you tell the compiler that a function is a near one, it will use a near call regardless of the memory model. To do this you must prototype it as a near function.

The syntax is:

```
int near NearFunc(int);
/* prototype for NearFunc */

int near NearFunc(int Number)
{
    /* statements */
}
```

A function that is in a different segment from the calling function must be invoked through a far call. Such a function must be prototyped and defined using the far keyword when the memory model is small or compact. For example:

```
int far FarFunc(int);
/* prototype for FarFunc */

int far FarFunc(int Number)
{
    /* statements */
}
```

You can declare function pointers using near and far keywords to produce 16- and 32-bit pointers, respectively. For example:

```
int (near *nFuncPtr) (int);
int (far *fFuncPtr) (int);
```

If you prototype a function with a formal parameter of far pointer type, and actually pass a near pointer, C automatically converts the near pointer to a far pointer. You should, however, use a type cast to document the conversion.

Modula-2 and Pascal possess no equivalent construct, although the same effect may be achieved by using pragmas. See Chapter : ‘*Pragmas*’ for further details.

Pointer Pitfalls

In both C/C++ and Modula-2/Pascal you should always take care when doing pointer arithmetic since there are several pitfalls.

When you add or subtract from a pointer, you must ensure that the value of its offset part does not exceed 65535 or go lower than 0. If either of these values is exceeded *segment wrap-around* results and this creates ‘difficult-to-find’ bugs. For example, if you add 3 to 65535 (the largest amount that can

be held in a 16-bit register), the result will be 2 and not 65538. The segment part of the pointer will not change.

Far pointer comparison may cause problems after pointer arithmetic, if you have two pointers derived in two entirely different ways, so that their segment parts are not the same. This is because two far pointers can point to the same location in memory, yet appear to have different addresses.

To illustrate the second point consider the segment: offset pairs:

```
0B87:0000
0B86:0010
```

They both point to the same address (under DOS). In spite of this, if you compare them using

```
if (ptr1 == ptr2)
```

or, in Modula-2:

```
IF (ptr1 = ptr2) THEN
```

the result would be false. This is because the two pointers are compared as if they were 32-bit integers.

A further problem arises if you use operators such as < or >. Only the offset part of the address is used to make the comparison.

In protected mode these problems become more complex still, because the segment register is itself only an index into a memory table.

Huge Pointers in C and C++

A *huge pointer* is declared using the keyword *huge* in the same way as the *near* and *far* keywords. For example:

```
char huge *cp;
```

declares a huge pointer.

No memory model allows static data objects greater than 64Kb. However, you can allocate data objects greater than 64Kb with the `halloc(size)` function. These objects must be accessed through huge pointers. For example:

```
char huge *hptr;
hptr = halloc(70000);
*hptr = 24;
hptr++;
```

A more complete example is:

```

#include <stdio.h>
#include <malloc.h>

long huge *ptr;
long huge *ptr2;

main()
{
    /* creates & prints a big 10 times table */
    long i;
    ptr = (long huge *) malloc (66000);
    if (ptr == NULL)
        puts("Can't allocate memory");
    else
    {
        ptr2 = ptr;
        for (i = 0; i < 16500; i++, ptr++)
            *ptr = i * 10;
        ptr = ptr2;
        for (i = 0; i < 16500; i++)
            printf ("%ld\n", *ptr++);
    }
}

```

Note: huge pointers may only be used to point at objects or arrays of objects where their size is a power of two (i.e. 1, 2, 4, 8, 16, 32, 64 bytes etc.). This is because TopSpeed does not normalize huge pointers, so that objects of other sizes might straddle a segment boundary.

The global variable `_hugeshift` is available for incrementing or decrementing a huge pointer's segment value. Using this variable guarantees portability between DOS and OS/2.

Pointing to Absolute Addresses

In a DOS program, it is possible to initialize a far pointer to point to an absolute address. The following example program demonstrates how you can initialize a pointer to point to the PC's monochrome memory buffer at segment address 0xB0000, offset 0.

```

/* program to print stars on the screen */

main()
{
    int i;
    char far *ptr++ = (char far *) 0xB0000000;
    for (i = 0; i < 2000; i++)
    {
        *ptr++ = '*';
        *ptr++ = 0x87; /* screen attribute*/
    }
}

```

In Modula-2 and Pascal (under DOS only) variables can be declared at a specified address:

```

MODULE MonoMem;

VAR
  ScrMem [0B000H:0000H]:
    ARRAY [1..25] OF
    ARRAY [1..80] OF
      RECORD
        Chr : CHAR;
        Atr : SHORTCARD;
      END;

```

or, in Pascal,

```

PROGRAM MonoMem;

VAR

  ScrMem [0B000H*16+0000H]:
    ARRAY [1..25] OF
    ARRAY [1..80] OF
      RECORD
        Chr : CHAR;
        Atr : byte;
      END;

```

Another way of writing the pointer declaration is to use a relative pointer. This forces a pointer to use a given variable to preload the segment register. For example, in C or C++:

```
unsigned seg = 0xB000; char <seg> *ptr = 0;
```

instead of:

```
char far *ptr = (char far *) 0xB0000000;
```

This new syntax means that ptr is a near pointer but using segment seg instead of DS to calculate its address. TopSpeed C will load the segment register from seg each time you dereference ptr, so seg can be updated if necessary. A similar syntax exists in Modula-2 and Pascal Æ see the relevant “*TopSpeed Language Reference Manual*”.

Pragmas

Mixed-model programming has traditionally been achieved in C using the keywords near and far. These keywords are not in the ANSI C standard and are not portable to other environments, such as UNIX. Pragmas provide a more portable method of using mixed models as well as providing greater control over calling and segmentation. A full list of pragmas is given in Chapter : ‘*Pragmas*’.

You can use the data pragma to determine the size of data pointers and the location of data. For example, you can place a data object in the default data segment like this:

```

#pragma save
#pragma data(seg_name => null)
int NearArray[17000];
#pragma restore

```

or in Modula-2:

```
(*# save *)
(*# data(seg_name => null) *)
VAR
  NearArray : ARRAY [1..17000] OF INTEGER;
(*# restore *)
```

The pragmas save and restore localize the effects of the seg_name pragma. The data pragma allows you to select the segment for the object. Selecting null for the segment name places NearArray in the default data segment _DATA.

You can place a data object in a separate segment:

```
#pragma save
#pragma data(seg_name => FARARRAY)
int FarArray[1000];
#pragma restore
```

This places the data object in a separate segment called FARARRAY_DATA.

You can also declare data pointers to be near or far using pragmas. The following example shows you how to declare a near pointer:

```
#pragma save
#pragma data(near_ptr => on)
int *NearPtr;
#pragma restore
```

or in Modula-2:

```
(*# save *)
(*# data(near_ptr => on) *)
VAR
  NearPtr : POINTER TO INTEGER;
(*# restore *)
```

This example shows you how to declare a far pointer:

```
#pragma save
#pragma data(near_ptr => off)
int *FarPtr;
#pragma restore
```

or in Modula-2:

```
(*# save *)
(*# data(near_ptr => off) *)
VAR
  FarPtr : POINTER TO INTEGER;
(*# restore *)
```

You can use the call pragma to determine whether a function should be called using a near or a far call. The following example shows you how to declare a near function:

```
#pragma save
#pragma call(seg_name => null, near_call => on)
int NearFunc(int);
#pragma restore
```

or in Modula-2:

```
(*# save *)
(*# call(seg_name => null, near_call => on) *)
PROCEDURE NearProc( i : INTEGER ) : INTEGER;
(*# restore *)
```

The call pragma has two parts. The first part, `seg_name`, tells TopSpeed to use the default code segment, `_TEXT`. The second part forces `NearFunc` to be called near.

The following example shows you how to declare a far function:

```
#pragma save
#pragma call(seg_name=>FARSEG, near_call=>off)
int FarFunc(int);
#pragma restore
```

The `seg_name` part of the call pragma places the function `FarFunc` in a code segment named `FARSEG_TEXT`. TopSpeed C calls `FarFunc` using far calls because `near_call` is off.

You can also determine the size of function pointers using the call pragma. For example, to declare a near function pointer:

```
#pragma save
#pragma call(near_call => on)
int (* nFuncPtr) (int);
#pragma restore
```

This example shows you how to declare a far pointer:

```
#pragma save
#pragma call(near_call => off)
int (* nFuncPtr) (int);
#pragma restore
```

You can use the call pragma in Modula-2 and Pascal to determine whether a procedure should be called using a near or a far call. The following example shows you how to declare a near procedure:

```
(*# save *)
(*# call(seg_name => null, near_call => on) *)

PROCEDURE NearProc( i : INTEGER ) : INTEGER;

(*# restore *)
```

A near function does not necessarily have to be in the segment named `_TEXT`, but it must be in the same segment as the calling function.

To declare a far function pointer:

```
#pragma save
#pragma call(near_call => off)
int (* fFuncPtr) (int);
#pragma restore
```

or:

```

(*# save *)
(*# call(seg_name => FARSEG, near_call => off)*)
VAR
  PROCEDURE FarProc( i : INTEGER ) : INTEGER;
(*# restore *)

```

You should be careful when prototyping functions with different pointer sizes using pragmas. For example, using keywords `near` and `far` you could declare:

```
int near NearFunc(void (far *FParm) (void));
```

However, the call pragma with the `near_call` parameter affects both the type of call used for the function *and* the size of any parameters which are themselves pointers to functions. This means that C will call `FParm` with a `near` rather than a `far` call. To achieve control over the parameter pointer sizes, types should be predefined for the parameters. For example:

```

#pragma save
#pragma call(near_call => off)
typedef void (far *FARFNCPTR) (void);
#pragma call(near_call => on)
int NearFunc(FARFNCPTR FParm);
#pragma restore

```

APPENDIX B

Class Object Formats

This appendix provides detailed and highly technical documentation on the layout of TopSpeed class objects. The information below is not normally required to write programs using TopSpeed Modula-2/Pascal object-oriented extensions or C++.

Conventions

In the following description, object means an instantiation of a class.

In layout examples, dw represents a 16-bit value; dd represents a 32-bit value.

Pascal/Modula-2

Method Tables

The method table pointer in an object points to a sub-descriptor in a single class descriptor for the whole class. This descriptor is unique for the program.

For each subclass there will be a reference point, and therefore there will be information relating to the subclass with both negative and positive offsets, relative to that reference point.

The method table pointers which are part of objects will be short pointers based on DS, except in the xlarge, mthread, overlay and dynalink models.

The method table pointer in an object is always present, and is the first field in the object. This is recursively true if the class contains multiple inherited classes.

The leftmost base class in a derived class will share object record fields and method pointer with the derived class.

Debug Information

Debug information relating to the whole class will be associated with the class descriptor.

Method Pointers

Pointers to methods are allocated with a positive offset relative to the reference point, starting from offset 0.

They will be allocated sequentially in the order of occurrence in the source. The pointers are 16-bit if the procedures are to be called with a near call, otherwise they will be 32-bit.

When a method is overridden in a derived class, the slot already allocated in the sub-class will be used for all accesses. The method will expect the self pointer to point to the sub-class.

Object Layout

At offset minus 2 bytes the size of the actual object is stored.

At offset minus 4 bytes the depth of the type hierarchy is stored.

Starting at offset minus 6 bytes and backwards, a table containing the type hierarchy is represented. This table has a default minimum number of entries to speed up access. The default minimum depth is currently 8 levels.

For each sub-class the topmost entry in the table (the most negative offset) will contain a pointer to the whole class. The rest of the table will contain nil values or a pointer to a parent class, in accordance with the depth of the class hierarchy.

The pointers in the class hierarchy are 16-bit in all models except for xlarge, mthread, overlay and dynalink, where they are 32-bit.

The sub-tables will be arranged according to the order which they occur in the source. Derived classes will share layout with the leftmost class.

For example, in xlarge model:

```

program e;

class x ;
  fx:int16;
  procedure px;
end;

class y inherits x ;
  fy:int16;
  procedure py;
end;

class x2;
  fx2:int16;
  procedure px2;
end;

class y2 inherits x2;
  fy2:int16;
  procedure py2;
end;

class z inherits x,y2;
  fz:int16;
  procedure pz;
  procedure px2;
end;

```

produces the following object layout:

```

public E@X:
  dd 0,0,0,0,0,0,0
  dd E@X
  dw 001H - depth
  dw 006H - object size
$1: dd E$X$PX

public E@Y:
  dd 0,0,0,0,0,0,0
  dd E@Y
  dd E@X
  dw 002H
  dw 008H
$2: dd E$X$PX
  dd E$Y$PY

public E@X2:
  dd 0,0,0,0,0,0,0
  dd E@X2
  dw 001H
  dw 006H
$3: dd E$X2$PX2

public E@Y2:
  dd 0,0,0,0,0,0,0
  dd E@Y2
  dd E@X2
  dw 002H
  dw 008H
$4: dd E$X2$PX2
  dd E$Y2$PY2

```

```

public E@Z:
  dd 0,0,0,0,0
  dd E@Z
  dd 000H
  dd E@X
  dw 003H
  dw 010H
$5: dd E$X$PX
     dd E$Z$PZ

sub table for y2

  dd 0,0,0,0,0
  dd E@Z
  dd E@Y2
  dd E@X2
  dw 003H
  dw 008H
$6: dd E$Z$PX2
     dd E$Y2$PY2

```

C++ to Pascal/Modula-2 Interface

This version will be as described above except with the following changes:

- **No information for the type hierarchy is stored. The type hierarchy is necessary for guard constraints to be checked, and this type test can not work without it. This is controlled by the data pragma `class_hierarchy => on | off`.**
- **When a method is overridden in a derived class, an extra entry for the method will be inserted in the first sub-table, except if there already is an entry for the method in the sub-table; in this case the old one will be used. The entry will point to a stub which will adjust the self pointer. This is controlled by the data pragma `cpp_compatible_class => on | off`.**

For example, in xlarge model:

```

program e;

class x ;
  fx:int16;
  procedure px;
end;

class y inherits x;
  fy:int16;
  procedure py;
end;

class x2 ;
  fx2:int16;
  procedure px2;
end;

```

```

class y2 inherits x2 ;
  fy2:int16;
  procedure py2;
end;

class z inherits x,y2;
  fz:int16;
  procedure pz;
  procedure px2;
end;

```

will produce the following object layout:

```

public E@X:
  dw 006H — object size
$1: dd E$X$PX

public E@Y:
  dw 008H
$2: dd E$X$PX
  dd E$Y$PY

public E@X2:
  dw 006H
$3: dd E$X2$PX2

public E@Y2:
  dw 008H
$4: dd E$X2$PX2
  dd E$Y2$PY2

public E@Z:
  dw 010H
$5: dd E$X$PX
  dd E$Z$PZ
  dd STUB( E$Z$PX2,+8)

```

sub table for y2:

```

  dw 008H
$6: dd E$Z$PX2
  dd E$Y2$PY2

```

C++

The C++ compiler uses the same representation as the Pascal and Modula-2 compilers with pragmas `data(class_hierarchy=>off,`
`ocpp_compatible_class=>on)`, except:

- The method table pointers in objects are only present if the class contains virtual methods or if the data pragma `compatible_class` is on.
- The position of the stubs and the entry for the method itself are reversed. Also the method expects a pointer to the class in which it is defined.
- Virtual base classes use a layout unique to C++.

Virtual Base Class Layout

A derived class containing virtual base classes has the following structure, starting from low memory:

1. First, non-virtual base classes appear in the order in which they were declared.
2. Next, pointers to virtual base classes appear in the order in which they were declared. In xlarge, mthread, overlay and dynalink models the pointers are 32-bit. In all other models they are 16-bit.
3. Lastly, the derived class itself appears.

Virtual Function Table Pointer

- In compatibility mode, the vptr behaves as in the Modula-2/Pascal implementations.
- When in non-compatibility mode, the vptr is the last member of the class.
- Left base classes share the vptr with the derived class.

Index

Symbols

/a option 38
 /b option 38
 /cc option 38
 /d option 38
 /e option 38
 /f option 38
 /j option 38
 /m option 38
 /n option 39
 /o option 39
 /op option 39
 /oq option 39
 /p option 39
 /pc option 39
 /pl option 39
 /r option 39
 /rg option 39
 /ri option 40
 /rn option 40
 /ro option 40
 /rr option 40
 /rs option 40
 /rv option 40
 /s option 40
 /t option 40
 /u option 40
 /v option 40
 /w option 40
 /zq option 38

A

abort Project System command 26
 action Project System macro 32
 alias optimize pragma 60
 and project operator 22
 array index checking
 pragma system 56
 array index checking:command line 40
 autocompile Project System command 18

B

batch interface 14
 boolean expressions in project files 22, 23

C

C Project System macro 32
 c_conv call pragma 45
 c_far_ext data pragma 52
 call pragmas
 c_conv 45
 ds_entry 47
 inline 46
 inline_max 50
 interrupt 46
 iopl 48
 near_call 45
 o_a_copy 48
 o_a_size 48
 opt_var_arg 51
 overlay 49
 reg_param 47
 reg_return 49
 reg_saved 48
 result_optional 49
 s_copy 50
 same_ds 45
 seg_jump 49
 seg_name 46, 48
 standard_conv 51
 standard_float 50
 t_l_copy 50
 t_l_size) 50
 var_arg 48
 var_str 50
 windows 49
 calling conventions
 command line 38
 pragma system 45, 50, 51
 cdir Project System macro 28, 32
 cgraph Project System macro 32
 check pragmas
 field 57
 guard 56
 index 56
 nil_ptr 56
 overflow 56
 range 56
 stack 56
 class compatibility
 pragma system 55
 class hierarchy 54
 command line
 /a option 38
 /b option 38
 /cc option 38

- /d option 38
- /e option 38
- /f option 38
- /j option 38
- /m option 38
- /n option 39
- /o option 39
- /op option 39
- /oq option 39
- /p option 39
- /pc option 39
- /pl option 39
- /r option 39
- /rg option 39
- /ri option 40
- /rn option 40
- /ro option 40
- /rr option 40
- /rs option 40
- /rv option 40
- /s option 40
- /t option 40
- /u option 40
- /v option 40
- /w option 40
- /zq option 38
- ansi keywords 38
- array index check 40
- calling convention 38
- constants in code 38
- coprocessor 39
- default char type 38
- define macro 38
- generate line numbers 38
- guard check 39
- language extensions 38
- memory models 38
- minimize blank lines in preprocessor 39
- nested comments 39
- nil pointer check 40
- optimizations 39
- overflow check 40
- preprocessor output 39
- quiet mode 38
- range check 40
- remake all 39
- retain comments in preprocessor outpu 39
- select processor 39
- set project macro 40
- set target operating system 40
- stack overflow check 40
- undefine macro 40

- variant record check 40
- vid debug level 40
- warnings level 40
- comparison project operator 22
- compatible_class data pragma 55
- compile mode 13
- compile Project System command 12, 16
- compile_src Project System macro 32
- const optimize pragma 58
- const_assign data pragma 55
- const_in_code data pragma 54
- constants in code
 - command line 38
 - pragma system 54
- convention
 - pragma system 49
- copro optimize pragma 60
- coprocessor
 - command line 39
 - pragma system 60
- cpath Project System macro 28, 32
- cpp_compatible_class data pragma 55
- cpu optimize pragma 60
- cse optimize pragma 58
- cwindow Project System macro 32

D

- data pragmas
 - c_far_ext 52
 - class_hierarchy 54
 - compatible_class 55
 - const_assign 55
 - const_in_code 54
 - cpp_compatible_class 55
 - ds_dynamic 54
 - ext_record 52
 - far_ext 51
 - heap_size 53
 - near_ptr 52
 - packed 54
 - seg_name 51
 - share_global 54
 - ss_in_dgroup 54
 - stack_size 53
 - threshold 55
 - var_enum_size 53
 - volatile 52
 - volatile_variant 52
- data segment
 - pragma system 45, 47, 54
- data threshold

- pragma system 55
- debug pragmas
 - proc_trace 61
 - vid 60
- debugging 12
 - pragma system 60, 61
- declare_compiler Project System command 29
- default char type
 - command line 38
- define macro
 - command line 38
- devsys Project System macro 33
- DLLs
 - dynamic linking 91
 - late binding 91
 - shared data and code 91
- dolink Project System command 18
- ds_dynamic data pragma 54
- ds_entry call pragma 47
- ds_eq_ss call pragma 48

E

- edit loadwin Project System command 25
- edit open Project System command 25
- edit refresh Project System command 25
- edit save Project System command 25
- edit saveall Project System command 25
- edit savewin Project System command 25
- editfile Project System macro 33
- editing project files 14
- editor commands in project 24
- editwin Project System macro; 33
- else Project System command 21
- elseif Project System command 21
- endif Project System command 21
- enumeration types
 - pragma system 53
- error format in Project System 24
- error Project System command 26
- errors Project System macro 24, 30, 32
- errors Project System macros 33
- exists project operator 22
- expand Project System command 28
- expr(promote) pragma 57
- expression promotion
 - pragma system 57
- ext Project System macro 28, 29, 33
- ext_record data pragma 52
- external variables
 - pragma system 52
- external variables:pragma system 51

F

- far_ext data pragma 51
- field check pragma 57
- file adderrors Project System command 24
- file append Project System command 23
- file copy Project System command 23
- file delete Project System command 23
- file move Project System command 23
- file prompt project system command 24
- file redirect Project System command 24
- file system commands in projects 23
- file touch Project System command 23
- filetype Project System macro 33
- function result
 - pragma system 49

G

- generate line numbers
 - command line 38
- getkey Project System command 27
- guard check
 - command line 39
 - pragma 56

H

- heap_size data pragma 53

I

- if Project System command 21
- ignore Project System command 21
- implib Project System command 19
- include Project System command 28
- index check pragma 56
- inline call pragma 46
- inline functions
 - pragma system 46
- inline_max call pragma 50
- interrupt call pragma 46
- interrupt functions
 - pragma system 46
- invoking projects 13, 14
- IO privilege
 - pragma system 48
- iopl call pragma 48

J

- jpicall Project System macro 33
- jump optimize pragma 59

L

- language extensions
 - command line 38
- Late binding 91
- link list 17
- link mode 14
- link Project System command 16, 18
- link_arg Project System macro 33
- linkage names
 - pragma system 57, 58
- linking 32
- long jumps
 - pragma system 49
- loop optimize pragma 59

M

- M Project System macro 33
- macro expansion in project 15
- main Project System macro 33
- make mode 13
- memory models 31
 - command line 38
- message Project System command 26
- minimize blank lines in preprocessor output:comman 39
- model Project System command 20

N

- name pragmas
 - prefix 57, 58
 - upper_case 57
- name Project System macro 29
- near calls
 - pragma system 45
- near heap size
 - pragma system 53
- near pointers
 - pragma system 52
- near_call call pragma 45
- near_ptr data pragma 52
- nested comments
 - command line 39
- nil pointer check
 - command line 40
- nil pointer checking
 - pragma system 56
- nil_ptr check pragma 56
- noedit Project System command 14
- not project operator 22

O

- o_a_copy call pragma 48
- o_a_size call pragma 48
- obj Project System macro 29
- odir Project System macro 28
- old compiler directives 43
- older project operator 22
- opath Project System macro 28
- open array parameters
 - pragma system 48
- opt_var_arg call pragma 51
- optimization
 - pragma system 58, 59, 60
- optimizations
 - command line 39
- optimize pragmas
 - alias 60
 - const 58
 - copro 60
 - cpu 60
 - cse 58
 - jump 59
 - loop 59
 - peep_hole 59
 - regass 59
 - speed 58
 - stk_frame 59
- or project operator 22
- overflow checking
 - command line 40
 - pragma system 56
- overlay call pragma 49
- overlay model
 - pragma system 49

P

- packed data pragma 54
- peep_hole optimize pragma 59
- pragma Project System command 21
- pragma syntax 42
 - C and C++ 44
 - Modula-2 43
 - Pascal 44
 - Project System 44
- pragma system
 - call class 45
 - check class 56
 - data class 51
 - debug class 60
 - expr class 57

- name class 57, 58
- optimize class 58
- predefined compilers 30
- prefix name pragma 57, 58
- preprocessor output
 - command line 39
- proc_trace debug pragma 61
- processor
 - pragma system 60
- project language 15
- Project System
 - #abort 26
 - #and 22
 - #autocompile 18
 - #compile 12, 16, 17
 - #declare_compiler 29
 - #dmlink 18
 - #edit loadwin 25
 - #edit open 25
 - #edit refresh 25
 - #edit save 25
 - #edit saveall 25
 - #edit savewin 25
 - #else 21
 - #elseif 21
 - #endif 21
 - #exists 22
 - #expand 28
 - #file adderrors 24
 - #file append 23
 - #file copy 23
 - #file delete 23
 - #file move 23
 - #file prompt 24
 - #file redirect 24
 - #file touch 23
 - #getkey 27
 - #if 21
 - #ignore 21
 - #implib 19
 - #include 28
 - #link 12, 16, 18
 - #message 26
 - #model 20
 - #noedit 14
 - #not 22
 - #older 22
 - #or 22
 - #pragma 21
 - #prompt 26
 - #run 27
 - #run-fail_abort 27
 - #run-no_abort 27
 - #run-no_window 27
 - #run-pause 27
 - #run-rte_abort 27
 - #run-session 27
 - #run-swap 27
 - #run-timed 27
 - #scan 29
 - #set 16
 - #split 28
 - #system 11, 20
 - action 32
 - batch interface 14
 - boolean expressions 22, 23
 - C 32
 - cdir 28, 32
 - cgraph 32
 - comparison operator 22
 - compile mode 13
 - compile_src 32
 - control flow 21
 - cpath 28, 32
 - cwindow 32
 - devsys 32
 - editfile 33
 - editing project files 14
 - editor commands 24
 - editwin 33
 - error format 24
 - errors 24, 30, 33
 - ext 28, 29, 33
 - file system commands 23
 - filetype 33
 - invoking a project:compile mode 13
 - invoking a project:make mode 13
 - jpical 33
 - language 15
 - link list 17
 - link mode 14
 - link_arg 33
 - linking 32
 - M 33
 - macro expansion 15
 - main 33
 - make mode 13
 - memory models 31
 - name 29
 - obj 29
 - odir 28
 - opath 28
 - predefined compilers 30
 - project file 11

- reconfiguration 30
- reply 24, 26, 27
- special project macros 32
- src 29
- tail 28
- unnamed.pr 13
- warnings 24, 30
- promote expr pragma 57
- prompt Project System command 26

R

- range checking
 - command line 40
 - pragma system 56
- reconfiguring Project System 30
- record field checking
 - pragma system 57
- record fields
 - pragma system 54
- reg_param call pragma 47
- reg_return call pragma 49
- reg_saved call pragma 48
- regass optimize pragma 59
- register parameters
 - pragma system 47
- register preservation
 - pragma system 48
- remake all
 - command line 39
- reply project macro 24
- reply Project System macro 26, 27
- result_optional call pragma 49
- retain comments in preprocessor output:command lin 39
- return value
 - pragma system 49
- run Project System command 27
- rundll Project System command 29

S

- s_copy call pragma 50
- same_ds call pragma 45
- scan Project System command 29
- seg_name call pragma 46
- seg_name data pragma 51
- segment naming
 - pragma system 46, 48, 51
- select processor
 - command line 39
- set Project System command 16
- set target operating system

- command line 40
- set_jump call pragma 49
- share_global data pragma 54
- smart linking 9
- special project macros 32
- speed optimize pragma 58
- split Project System command 28
- src Project System macro 29
- ss_in dgroup pragma 54
- stack check pragma 56
- stack checking:pragma system 56
- stack location
 - pragma system 54
- stack size
 - pragma system 53
- stack_size data pragma 53
- standard_conv call pragma 51
- standard_float call pragma 50
- static linking 91
- stk_frame optimize pragma 59
- string copying
 - pragma system 50
- structured constants
 - pragma system 55
- system Project System command 11, 20

T

- t_l_copy call pragma 50
- t_l_size call pragma 50
- tail Project System macro 28
- threshold data pragma 55
- type checking
 - pragma system 50
- type transfer guard checking:pragma system 56
- type-safe linking 10, 11
- typeless parameters
 - pragma system 50
- typographic conventions
 - general 8

U

- undefine macro
 - command line 40
- unnamed.pr 13
- upper_case name pragma 57

V

- var_arg call pragma 48
- var_enum_size data pragma 53
- var_str call pragma 50

- variable argument list
 - pragma system 48, 51
- variant records
 - command line 40
 - pragma system 52
- vid debug level
 - command line 40
- vid debug pragma 60
- volatile data pragma 52
- volatile variables
 - pragma system 52
- volatile_variant data pragma 52

W

- warnings level
 - command line 40
- warnings Project System macro 24, 30
- windows call pragma 49