TopSpeed®

For IBM® Personal Computers and Compatibles

Advanced Programmer's Guide

TopSpeed Corporation

Printed in the United Kingdom.
10 9 8 7 6 5 4 3 2 1

Contents

Chapter 1 Introduction

This manual describes the programs, files and facilities comprising the TopSpeed TechKit© and provides advanced programming information for users of the TopSpeed Environment. The TechKit© offers tools for full, professional development with the TopSpeed Environment and any of the TopSpeed compiler suite:

- Supports large program and data management:
- A Segment-based Swapping Overlay Management System for DOS, manages programs and data up to 8MB in size.
- Supports Dynamic Link Libraries under both OS/2 and MS-DOS.
- Supports programming for Microsoft Windows.
- Multilanguage support:
- Documentation for cross-linking programs written in different languages.
- The TopSpeed Assembler: A full 80x86/80x87 assembly language, compatible with the TopSpeed multi-language suite.
- Development tools:
- Post-Mortem Debugging - allows you to debug aborted programs after they crash.
- Watch - allows you to monitor DOS function calls.
- Utility Programs: a Disassembler and a Program Profiler.
- Advanced technical information:
- TSR (Terminate and Stay Resident) support for TopSpeed Modula-2.
- A description of the procedures for configuring and using your computer's RS-232 port for TopSpeed Modula-2.
- Support for embedded systems: an explanation of library initialization procedures which you will need in order to run in a non-standard environment.

- Technical appendices supplementing information contained in the "TopSpeed Developer's Guide", and documenting many operating system and TopSpeed features you need to know about in order to write advanced applications.

Knowledge Requirements

The facilities provided by the TopSpeed TechKit© are directed at the advanced programmer. It is therefore assumed that you are familiar with DOS, and the programming language you are working in. A working knowledge of the TopSpeed Environment and the information contained in the "*TopSpeed Developer's Guide*" is also assumed.

No further knowledge or resources are assumed for most of the facilities described in this manual, except for localized knowledge of the area under consideration. For example, if you are using the *Watch* facility, we assume that you are already familiar with interrupts and their use.

The appendices of this manual provide technical information. They are included as a reference summarizing topics discussed within the main body of this guide.

You may find it useful to supplement your reading of one chapter by referring to other chapters and to the technical appendices. For example, a good understanding of DLLs will help you to understand how to develop your own overlays. Each chapter suggests any additional chapters which you may find helpful.

The rest of this chapter explains where to find the more important, specialized knowledge relevant to some TechKit© facilities.

Memory Models and Pointer Programming

The segmented address structure of the Intel 80*x*86 series of chips and their associated industry-standard computers is one of the most complex — and ultimately, perhaps, unnecessary — features, which distinguishes it from the simpler architectures of many other computers. The 80386 and 80486 chips, which, in principle, pave the way to a more reliable and simpler program structure, and the OS/2 and Windows environments, which can make full use of the virtual address space provided by these chips, have complicated rather than simplified the job of the programmer, who is obliged to support not only the new architecture, but many parts of the old one as well.

The net result for the non-Unix PC programmer is that the most difficult and technically demanding tasks are often associated not with the language, but with the machine and system-dependent problems of managing the 80*x*86's

tortuous address structure, and the complexities of the operating systems which sit on top of it.

A certain amount of informal standardization has been reached, if only due to:

- The basic programming standards imposed by DOS.
- The executable file format DOS demands.
- The object code format demanded by the Microsoft Linker.

Modern PC compilers have built on the conventions established by Microsoft, but have, by and large, outgrown it, so that different programming languages differ in many subtle ways in their implementation of memory models, in their organization of segments and in their treatment of the heap and the stack.

The information on memory models provided by the "*TopSpeed Developer's Guide*" is adequate for all but the most technical applications. However, more knowledge is required if you wish to implement your own management scheme, or write applications which interface with proprietary operating systems and environments — such as Windows. Appendix A: '*Memory models*' supplements the "*TopSpeed Developer's Guide*" with information which is useful, if not essential, for the OS/2 and Windows programmer, and serves as a reference for the conventions followed by JPI.

Calling Conventions

TopSpeed's system for passing parameters between functions makes highly efficient use of the 80*x*86 registers, and adds substantially to the speed and compactness of TopSpeed code.

Other language implementations, and operating systems to which your TopSpeed application must interface, use conventions which are different from each other and from TopSpeed.

Thus, if you are linking to third party library systems, or using operating system calls other than via the interface provided by the TopSpeed library, you must ensure that you use the stack and registers in a way which is acceptable to the client system.

It is essential that you read the section on '*The JPI calling convention*' in Chapter 5 if you are not familiar with the differences between the calling conventions expected by TopSpeed, and those expected by other systems.

Dynamic Link Libraries and the New Executable File Format

TopSpeed is unique in providing *Dynamic Link Libraries* for DOS users. This manual, therefore, provides the DOS user with a simple account of the philosophy and techniques associated with DLLs (see Chapter : '*Dynamic Link Libraries*'). If you are a DOS user and are unfamiliar with OS/2 programming concepts, this will equip you to produce professional DOS DLLs.

Note: This information is not a substitute for Microsoft's OS/2 documentation, and if you are an OS/2 user you will probably find it helpful to supplement your use of this manual by referring either to the OS/2 documentation or a good OS/2 programmer's guide.

This section is of general relevance to Chapter : '*Segment based-Overlays*', Chapter : '*Dynamic Link Libraries*' and Chapter : '*Windows programming*'. All these facilities use the form of executable file and associated conventions and utilities, known as the *New Executable File Format*. This format extends the executable format originally introduced with MS-DOS. The two most important extensions are:

- Provision of additional information to control segment reallocation in multi-thread systems, such as OS/2 and Windows.
- Provision of additional linkage information to control late (run-time) binding of separate segments of an application.

The new executable file format is associated with extra files, which supply information for linkers and loaders, and utilities for processing these files.

Appendix B: '*Module Definition File Syntax*' supplies information and guidance on the use of these facilities, supplementing the material in the chapters named above.Programming Examples

TopSpeed is a *multi-language environment* supporting four major languages: C, C++, Modula-2 and Pascal.

You do not need more than one language to use the TechKit$^{\hat{O}}$, but you will find some understanding of the conventions applied by other languages useful.

Where necessary, this manual supplies programming examples in two languages. However, for some short programming examples where the meaning is obvious and repetition would be tedious, this is not done.

The programming examples use Modula-2 and C. It is assumed that you have sufficient knowledge to convert these examples for Pascal or C++ applications.

Typographic Conventions

The following typographic conventions are used throughout this manual:

Italics	are used for emphasis and to introduce new concepts
Courier	is used for program examples and other items which would appear on the screen
small italics	are used in syntax descriptions

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

Alt-C

This means you should hold down the *Alt* key and press *C*.

Where example screen outputs are shown, the illustration is boxed.

CHAPTER 2
SEGMENT-BASED OVERLAYS

Introduction

TopSpeed's *Segment-based Overlay Management System* (OMS) allows you to write programs of virtually any size (up to 254 segments), which can be run on any DOS machine. This is achieved 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 frees 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, if required, it can operate entirely automatically. There is no requirement to specify an overlay structure: all you have to do is specify a special *overlay memory model*. However, should you want to control the overlay process in more detail, you can do so.

Controllable Overlays

The simplest way to gain a measure of control over the overlay process is to flag specified segments as *preloaded* or *discardable*. These flags are taken into account when decisions are made about which segments to swap in and out.

Full control is available through a range of *Overlay Management Procedures*, provided by the overlay *Application Programmers Interface* (API), which the program itself may invoke. This means that the selection of segments to be swapped or discarded can be performed by the application.

You can take control over which parts of the program are placed in which segment, by means of the call(seg_name)pragma described below. This can help to cut down on swapping to ensure the most efficient use of CPU time.

Overlays and Multi-thread Programs

While you are not generally 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 restrictions imposed by DOS make it impossible to guarantee that a segment is inactive under all conditions. Multi-thread programs must deal with this by marking segments used by child threads as resident, or by assuming full control of overlays. This is described in more detail later in this chapter.

OS/2 and Dynamic Linking

The TopSpeed 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*'), which facilitate code-sharing and more economical use of disk space.TopSpeed also provides Dynamic Link Libraries for DOS, which bring substantial benefits but do not, of course, facilitate code-sharing since DOS is not a multi-tasking system.

A Note for Windows Programmers

If you wish to develop applications for Microsoft Windows, you should note that Windows has its own memory management system and its own special form of Dynamic Link Library. If you are programming for Windows, use the techniques described in Chapter : '*Windows Programming*'.

Compiling and Running an Overlayed Program

Using the *TopSpeed Overlay System* is straightforward. All you need to do is specify a special *overlay memory model* for your application. This can be done in two ways from within the TopSpeed Environment, either:

- Edit the project file to include the statement:

```
#model overlay
```

or,

- Use the Project Memory model option from the environment menu to set the model.

The Project System automatically compiles and links 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 an *export* file with a .EXP

extension. When TopSpeed is installed, a default export file, OVERLAY.EXP, is provided. The default redirection files include the location of this file. You, therefore, do not need to take any special steps, except to ensure that this file is present and accessible through the redirection file.

As explained in the next section, you can create your own .EXP file to gain more control over the overlay process; this must have the same name as your project file and must be accessible via the redirection file.

For more information on export files see Appendix B: '*Module Definition File Syntax*'.

Running an Overlayed Program

Overlayed programs may be called just like an ordinary .EXE file. The only requirements to observe are:

- There must either be expanded memory or 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 that 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 when designing overlayed programs:

- 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 int

o the segment <name>. It can be used both to break a compilation unit into parts and to combine code from different compilation units.

- Whether segments should be labelled as possessing special properties, such as PRELOAD or DISCARDABLE. This is controlled by editing a .EXP file as explained below. Interrupt handlers, for example, should always be in resident segments.

- Whether to take direct control of the overlay process using the procedures described in this chapter. This will be necessary under two circumstances:
- If you need to optimize performance by taking advantage of the special knowledge that you have about the pattern of intersegment calls. The TopSpeed system, for example, knows that you can never edit and compile a program at the same time, and uses this knowledge to allocate memory more efficiently.
- If you find that you are running out of main memory, due to the granularity of your segment structure. (This can happen, for example, when the memory manager swaps out a small segment but needs to swap in a larger one). Even though a space has been cleared, it cannot be used because, in general, memory modules cannot be dynamically relocated in main memory.

Program and Data Layout

The basic unit of memory used by the TopSpeed Overlay System for swapping is the *segment*. Entire segments are swapped in and out. Thus, 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 default program layout in the overlay memory 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:

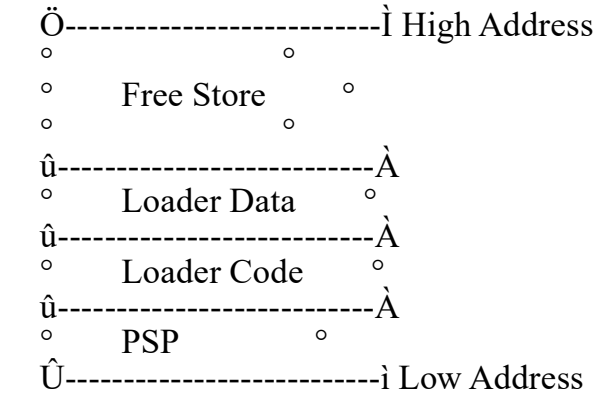
The basic principles of good segment layout are:

- 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.
- Use small modules to reduce granularity.
- Put all interrupt handlers in resident segments.
- Do not over-ride the default pragma settings same_ds or ds_entry.

The Default Memory Management Algorithm

TopSpeed contains a *Default Overlay Algorithm*, which is applied if no other course of action is taken. 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 memory is shared between the TopSpeed Overlay Manager and the far heap. Any library functions which 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 necessary using the following rules:

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

If insufficient memory is available to satisfy a request from user heap functions or from the TopSpeed Overlay Manager, inactive segments are unloaded until sufficient space is 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 fails. This is handled in the normal manner for the language involved. If the request was from the TopSpeed 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.

Expanded memory, if present, is used as a temporary store for CODE segments and to fixup frequently used segments.

All inactive segments may be unloaded explicitly by a call to the overlay API flush function.

Memory Available Functions

Functions that indicate the amount of memory in the far heap, for example:

```
Storage.AVAILABLE (Modula-2)
farcoreleft, coreleft (C)
```

can sometimes give misleading results in an overlayed program.

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 a allocation function fails, the TopSpeed Overlay Manager unloads 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.

A simple solution is to call UserFlush before calling Storage.AVAILABLE for example. This gives 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 "*Modula-2 Reference*")
C & C++	check for a NULL return value.
Pascal	install a heap error handler, returning 1. (See the '*Pascal Library Reference*')

Segment Characteristics and the .EXP File

The overlay algorithm can be influenced by providing extra information. You can specify a segment as;

- discardable
- preloaded.

This is done at link time, by creating a special file with the same name as your project, and the extension .EXP. This file must be accessible via the redirection file. If you do not create such a file, the linker will use the default file OVERLAY.EXP, which is supplied and installed with the TopSpeed Environment.

Finding Out Your Segment Name

In order to mark segments with special characteristics, you need to know the names which have been allotted to these segments. The compiler assigns names according to the conventions described in Appendix A. If in doubt, you can list the .MAP file produced by the linker, to find out the names which have been assigned to the segments you are interested in.

Preloaded and Discardable Segments

The syntax of the .EXP file is described below, and in more detail in Appendix B.

Preloaded Segments

Segments may be marked as PRELOAD. In this case, they will be loaded initially and will not be swapped out.

Discardable Segments

Demand-loaded DATA segments which are not required after use can be marked as DISCARDABLE. This improves program performance since the segments free swapfile space and other resources.

A good example is a procedure which requires a static buffer. If the buffer is always re-initialized on entry, there is no reason to preserve its contents when the procedure is inactive. The buffer's segment should be marked as DISCARDABLE.

Note: A segment must not be marked as both PRELOAD and DISCARDABLE.

Syntax of the .EXP File

The technique and defaults vary depending on whether the segment is a CODE segment or a DATA segment.

CODE Segments

The default .EXP file used for an overlayed program marks certain segments as PRELOAD. These are those library segments used by floating point, interrupt and exception handling code. These must not be changed and you should not use them. They are:

```
STACK
_INIT
CPROC_TEXT
PROCESS_TEXT
PROC_TEXT
PASPROC_TEXT
SIG_TEXT
EMU_TEXT
MP_CODE
_DATA
```

You may mark areas of your own code as resident for the following reasons:

- Any segment containing an interrupt handler should remain resident.
- Any segment containing frequently used code. (Although this kind of segment is less likely to be loaded, a saving in overhead can be made if the procedures involved are small).
- Any segment containing code used for handling loader out-of-memory errors.

The SEGMENTS statement in the following example has been added to the default .EXP file, and marks modules USER1_TEXT and USER2_TEXT as PRELOAD and, therefore, resident. These are the modules containing the code for modules USER1.MOD and USER2.MOD (Modula-2) or USER1.C(pp) and USER2.C(pp) (C/C++).

```
DATA     PRELOAD

SEGMENTS  STACK         PRELOAD
          _INIT         PRELOAD
          CPROC_TEXT    PRELOAD
          PROCESS_TEXT  PRELOAD
          PROC_TEXT     PRELOAD
          PASPROC_TEXT  PRELOAD
          SIG_TEXT      PRELOAD
          EMU_TEXT      PRELOAD
          MP_CODE       PRELOAD
          _DATA         PRELOAD

SEGMENTS
          USER1_TEXT    PRELOAD
          USER2_TEXT    PRELOAD
```

DATA Segments

The default .EXP file marks all data as PRELOAD. However, DATA segments may be loaded on demand safely, as long as the addresses of objects within them are not stored in static data. VAR parameters and pointer parameters may be used. In general, it is safe to use swappable DATA segments in Modula-2 and Pascal, but not in C and C++.

A moveable (swappable) DATA segment will be loaded when first referenced by a CODE segment, and swapped to disk when no active CODE segments reference it. If the data is discardable, performance can be improved by marking the segment as DISCARDABLE.

Note: If a DATA segment is unloaded after a call to UnLoadModule, its contents are not swapped to disk, and if the module is reloaded its contents are re-initialized.

For example:

```
DATA        LOADONCALL    MOVEABLE
SEGMENTS    STACK         PRELOD

  _INIT         PRELOAD
 CPROC_TEXT     PRELOAD
 PROCESS_TEXT   PRELOAD
 PROC_TEXT      PRELOAD
 PASPROC_TEXT   PRELOAD
  SIG_TEXT      PRELOAD
          EMU_TEXT      PRELOAD
          MP_CODE       PRELOAD
          _DATA         PRELOAD
```

All DATA segments, apart from the library segment _DATA, are now demand-loaded. _DATA must always be marked PRELOAD.

Programming for the Overlay Model

The *overlay model* is a superset of the *multi-thread model*, which in turn is a subset 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 that it provides separate DATA and CODE segments for each compilable module, giving you 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 which is referenced by one thread is not swapped out by another.

Overlay System API

The following library functions are provided to allow direct control over the overlay process.

Note: In the programming interface below, the parameter SegNo refers to the segment number, which is obtained from the .MAP file produced by the linker. The ModName parameter refers not to the name of the compiled source unit but to the .DLL file — a slight confusion of terminology.

C/C++ Language Bindings

The header file overlay.h contains the declarations of all functions and types. The Project System will ensure that the correct library containing these functions is linked.

void SetExitHandler(ExitHandler P);

A function may be installed to enable error handling, allowing you to manage overlay load failure. The function P is established as the exit handler to be called, when the TopSpeed Overlay Manager generates an exception condition.

`P` is invoked when the overlay handler encounters an error. The name of the overlay program/module and an error code are passed to the error handler. The error codes are as follows:

Error code	Number
LOADER_ERROR_INVALID_ENTRY	1
LOADER_ERROR_TOO_MANY_MODULES	2
LOADER_ERROR_WRONG_MODULE_VERSION	3
LOADER_ERROR_MODULE_CORRUPT	4
LOADER_ERROR_STACK_CORRUPT	5
LOADER_ERROR_MODULE_NOT_FOUND	6
LOADER_ERROR_FIXUP_INCORRECT	7
LOADER_ERROR_EMS_ERROR	8
LOADER_ERROR_NOT_DLL	9
LOADER_ERROR_STACK_TRACE	10
LOADER_ERROR_MODULE_INIT_FAILED	11
LOADER_ERROR_FILE_NOT_FOUND	12
LOADER_ERROR_OUT_OF_MEMORY	13
LOADER_ERROR_FILE_READ	14
LOADER_ERROR_FILE_WRITE	15

LOADER_ERROR_STACK_OVERFLOW	16
LOADER_ERROR_CODE_FIXUP	17
LOADER_ERROR_TOO_MANY_SEGMENTS	18
LOADER_ERROR_INVALID_LOAD	19
LOADER_ERROR_CMEM_ERROR	20
LOADER_ERROR_INVALID_IMPORT	21
LOADER_ERROR_NO_ENTRYPOINTS	22
LOADER_ERROR_ILLEGAL_TRANSFORM	23
LOADER_ERROR_SWAP_FILE	24
LOADER_ERROR_ILLEGAL_ADDITIVE	25

The exit handler function should have the form:

```
void handler(char *name,unsigned code);
```

An exit handler should be in a resident segment (marked PRELOAD), and may not call functions in any non-resident segment.

If an error occurs in the TopSpeed Overlay Manager after a call to the exit handler function, the process will terminate immediately.

void SetMemHandler(MemHandler P);

This function installs an "out-of-memory" handler which is called when a memory allocation request fails. It is passed the allocation size in bytes. If the function returns non-zero, the request is retried.

Note: Error and out-of-memory handlers must reside in PRELOAD segments.

The out-of-memory handler function should have the form:

```
int handler(unsigned bytes);
```

void UserFlush(void);

The UserFlush function unloads all inactive overlays and DLLs.

unsigned LoadSeg(unsigned SegNo, char *ModName);

The LoadSeg function loads the specified segment from the specified module. NULL may be passed as the second parameter, signifying the main module. The return value indicates the status of the operation:

Return value		Meaning
0	LOADER_SUCCESS	Segment was loaded.

1	LOADER_FAIL	Segment could not be loaded.
2	LOADER_RESIDENT	Segment already loaded.
4	LOADER_INVALID_SEG	Segment number invalid.
5	LOADER_INVALID_MODULE	Module name invalid.

void UnloadSeg(unsigned SegNo, char *ModName);

The UnloadSeg function unloads the specified segment from the specified module. NULL may be passed as the second parameter signifying the main module. A simple program in the overlay model has only a main module.

The return value indicates the status of the operation:

Return value	Meaning
0 LOADER_SUCCESS	**Segment was unloaded.**
1 LOADER_FAIL	**Segment could not be unloaded.**
3 LOADER_UNLOADED	**Segment already unloaded.**
4 LOADER_INVALID_SEG	**Segment number invalid.**
5 LOADER_INVALID_MODULE	**Module name invalid.**

void SetMode(unsigned Mode);

Initializes overlay mode. The parameter may be LOADER_MANUAL or LOADER_AUTO. LOADER_AUTO is the default. If manual operation is selected, segments are NOT unloaded automatically. See '*Multi-thread Programming Using Overlays*', later in this chapter.

unsigned LoadModule (const char *ModName);
void UnLoadModule(const char *ModName);
void * GetProcAddr (unsigned ModNo, const char *ProcName);
void Reset(void)

LoadModule loads the specified module (DLL). A handle is returned. If the module was not found, 0xFFFF is returned.

GetProcAdr returns the address of the module specified by a handle returned from LoadModule.

UnLoadModule unloads the specified module.

Reset should be used after a longjmp from an "out-of-memory" handler. The loader is reset and all inactive segments freed.

For more information on using DOS DLLs see Chapter : '*Dynamic Link Libraries*'.

void Terminate(void);

This function calls the TopSpeed Overlay System shutdown procedures. This function is called by the standard library termination functions exit and _exit, so does not need to be called explicitly unless a non-standard program termination is used.

Modula-2 Language Bindings

The definition file, overlay.def, contains the definitions of all functions, procedures and types. The TopSpeed Project System ensures that the correct library containing these functions is linked.

Note: the str65 string type must be zero terminated.

PROCEDURE SetExitHandler(P: ExitHandler);

This procedure enables error handling, allowing you to manage an overlay load failure. The function P is established as the exit handler to be called when the TopSpeed Overlay Manager generates an exception condition.

The exit handler procedure should have the form:

```
PROCEDURE Handler(Name: ARRAY OF CHAR;
                  Code: CARDINAL);
```

The handler procedure is invoked when the overlay handler encounters an error. The name of the overlay and an error code are passed to the error handler. The error codes are as follows:

Error code	Number

```
LOADER_ERROR_INVALID_ENTRY              1
LOADER_ERROR_TOO_MANY_MODULES       2
LOADER_ERROR_WRONG_MODULE_VERSION   3
LOADER_ERROR_MODULE_CORRUPT             4
LOADER_ERROR_STACK_CORRUPT              5
LOADER_ERROR_MODULE_NOT_FOUND       6
LOADER_ERROR_FIXUP_INCORRECT            7
LOADER_ERROR_EMS_ERROR                  8
LOADER_ERROR_NOT_DLL                9
LOADER_ERROR_STACK_TRACE            10
LOADER_ERROR_MODULE_INIT_FAILED         11
LOADER_ERROR_FILE_NOT_FOUND             12
LOADER_ERROR_OUT_OF_MEMORY              13
LOADER_ERROR_FILE_READ                  14
LOADER_ERROR_FILE_WRITE                 15
LOADER_ERROR_STACK_OVERFLOW             16
LOADER_ERROR_CODE_FIXUP                 17
LOADER_ERROR_TOO_MANY_SEGMENTS      18
LOADER_ERROR_INVALID_LOAD           19
LOADER_ERROR_CMEM_ERROR                 20
LOADER_ERROR_INVALID_IMPORT             21
LOADER_ERROR_NO_ENTRYPOINTS             22
LOADER_ERROR_ILLEGAL_TRANSFORM      23
LOADER_ERROR_SWAP_FILE                  24
LOADER_ERROR_ILLEGAL_ADDITIVE       25
```

PROCEDURE SetMemHandler(P: MemHandler);

This procedure installs an out-of-memory handler which is called when a memory allocation request fails. It is passed the allocation size in bytes. If the function returns non-zero the request is retried.

Note **Error and out-of-memory handlers must be in PRELOAD segments.**

The out-of-memory handler procedure should have the form:

PROCEDURE Handler(Size: CARDINAL): CARDINAL;

PROCEDURE UserFlush();

The UserFlush() procedure unloads all inactive overlays and DLLs.

PROCEDURE LoadSeg(SegNo: CARDINAL; ModName: str65): CARDINAL;

The LoadSeg function loads the specified segment from the specified module. The constant MainModule may be passed as the second parameter, signifying the main module.

The return value indicates the status of the operation:

Return value		Meaning
0	LOADER_SUCCESS	**Segment was loaded.**
1	LOADER_FAIL	**Segment could not be loaded.**
2	LOADER_RESIDENT	**Segment already loaded.**
4	LOADER_INVALID_SEG	**Segment number invalid.**
5	LOADER_INVALID_MODULE	**Module name invalid.**

PROCEDURE UnloadSeg(SegNo: CARDINAL; ModName: str65): CARDINAL;

The unload segment function unloads the specified segment from the specified module. The constant MainModule may be passed as the second parameter signifying the main module.

The return value indicates the status of the operation:

Return value		Meaning
0	LOADER_SUCCESS	**Segment was unloaded.**
1	LOADER_FAIL	**Segment could not be unloaded.**
3	LOADER_UNLOADED	**Segment already unloaded.**
4	LOADER_INVALID_SEG	**Segment number invalid.**
5	LOADER_INVALID_MODULE	**Module name invalid.**

PROCEDURE SetMode(Mode: CARDINAL);

Initializes overlay mode. The parameter may be LOADER_MANUAL or LOADER_AUTO. LOADER_AUTO is the default. If manual operation is selected segments are NOT unloaded automatically. See '*Multi-thread Programming Using Overlays*' later in this chapter.

PROCEDURE LoadModule (ModName: str65): CARDINAL; PROCEDURE UnLoadModule(ModName: str65); PROCEDURE GetProcAddr (ModNo: CARDINAL; ProcName:

```
str65):ADDRESS;
```

PROCEDURE Reset();

LoadModule loads the named module (DLL). A handle is returned. If the module was not found MAX(CARDINAL) is returned.

GetProcAdr returns the address of the module specified by a handle returned from LoadModule.

UnLoadModule unloads the specified module.

Reset should be used after a longjmp from an out-of-memory handler. The loader is reset and all inactive segments freed.

For more information on using DOS DLLs see Chapter 3: '*Dynamic Link Libraries*'.

PROCEDURE Terminate();

The procedure Terminate() calls the TopSpeed Overlay System shutdown procedures. This function is called by HALT and does not need to be called by you unless a non-standard program termination is used.

Pascal Language Bindings

The interface file PasOvl.ITF contains the definitions of all functions, procedures and types. The TopSpeed Project System ensures that the correct library containing these functions is linked.

Note: the str65 string type must be zero terminated.

PROCEDURE SetExitHandler(P: ExitHandler);

This procedure enables error handling, allowing you to manage an overlay load failure. The function P is established as the exit handler to be called when the TopSpeed Overlay Manager generates an exception condition.

`P` is invoked when the overlay handler encounters an error. The error handler is passed the name of the overlay and an error code. The error codes are as follows:

Error code	Number

```
LOADER_ERROR_INVALID_ENTRY              1
LOADER_ERROR_TOO_MANY_MODULES       2
LOADER_ERROR_WRONG_MODULE_VERSION   3
LOADER_ERROR_MODULE_CORRUPT             4
LOADER_ERROR_STACK_CORRUPT              5
LOADER_ERROR_MODULE_NOT_FOUND       6
LOADER_ERROR_FIXUP_INCORRECT            7
LOADER_ERROR_EMS_ERROR                  8
LOADER_ERROR_NOT_DLL                9
LOADER_ERROR_STACK_TRACE            10
LOADER_ERROR_MODULE_INIT_FAILED         11
LOADER_ERROR_FILE_NOT_FOUND             12
LOADER_ERROR_OUT_OF_MEMORY              13
LOADER_ERROR_FILE_READ                  14
LOADER_ERROR_FILE_WRITE                 15
LOADER_ERROR_STACK_OVERFLOW             16
LOADER_ERROR_CODE_FIXUP                 17
LOADER_ERROR_TOO_MANY_SEGMENTS      18
LOADER_ERROR_INVALID_LOAD           19
LOADER_ERROR_CMEM_ERROR                 20
LOADER_ERROR_INVALID_IMPORT             21
LOADER_ERROR_NO_ENTRYPOINTS             22
LOADER_ERROR_ILLEGAL_TRANSFORM      23
LOADER_ERROR_SWAP_FILE                  24
LOADER_ERROR_ILLEGAL_ADDITIVE       25
```

The exit handler procedure should have the form:

```
procedure Handler(Name: str65; Code: word);PROCEDURE SetMemHandler(P:
MemHandler);
```

This procedure installs an out-of-memory handler which is called when a memory allocation request fails. It is passed the allocation size in bytes. If the function returns non-zero the request will be retried.

Note: Error and out-of-memory handlers must reside in PRELOAD segments.

The out-of-memory handler should have the form:

```
function Handler(Size: word): word;
```

PROCEDURE UserFlush;

The UserFlush procedure unloads all inactive overlays and DLLs.

FUNCTION LoadSeg(SegNo: word; ModName: str65): word;

The load segment function loads the specified segment from the specified module. An empty string may be passed as the second parameter, signifying the main module.

The return value indicates the status of the operation:

Return value		Meaning
0	LOADER_SUCCESS	Segment was loaded.
1	LOADER_FAIL	Segment could not be loaded.
2	LOADER_RESIDENT	Segment already loaded.
4	LOADER_INVALID_SEG	Segment number invalid.
5	LOADER_INVALID_MODULE	Module name invalid.

FUNCTION UnloadSeg(SegNo: word; ModName: str65): word;

The unload segment function unloads the specified segment from the specified module. A null string ‘’ may be passed as the second parameter, signifying the main module.

The return value indicates the status of the operation:

Return value		Meaning
0	LOADER_SUCCESS	Segment was unloaded.
1	LOADER_FAIL	Segment could not be unloaded.
3	LOADER_UNLOADED	Segment already unloaded.
4	LOADER_INVALID_SEG	Segment number invalid.
5	LOADER_INVALID_MODULE	Module name invalid.

PROCEDURE SetMode(Mode: word);

Initializes overlay mode. The parameter may be LOADER_MANUAL or LOADER_AUTO. LOADER_AUTO is the default. If manual operation is selected segments are NOT unloaded automatically. See ‘*Multi-thread Programming Using Overlays*’ later in this chapter.

```
FUNCTION LoadModule (ModName: str65): word;
  PROCEDURE UnLoadModule(ModName: str65);
  FUNCTION GetProcAddr (ModNo: word; ProcName: str65):
  address;
  PROCEDURE Reset;
```

LoadModule loads the specified module (DLL). A handle is returned. If the module was not found the value maxword is returned.

GetProcAddr returns the address of the module specified by a handle returned from LoadModule.

UnLoadModule unloads the specified module.

Reset should be used after a longjmp from an out-of-memory handler. The loader is reset and all inactive segments freed.

For more information on using DOS DLLs see Chapter 3: '*Dynamic Link Libraries*'.

PROCEDURE Terminate;

The procedure Terminate calls the TopSpeed Overlay System shutdown procedures. This function is called by HALT, and does not need to be called explicitly unless a non-standard program termination is used.

Run-time Errors

The following runtime errors may be reported by the loader:

LOADER_ERROR_INVALID_ENTRY (Code 1)

Invalid entry in the 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 JPI.

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 TopSpeed.

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 Chapter 6: '*TopSpeed Assembler*'.

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 TOPSPEED.

LOADER_ERROR_INVALID_IMPORT (Code 21)

Internal error.

Action: Report to TOPSPEED.

LOADER_ERROR_NO_ENTRYPOINTS (Code 22)

Internal error.

Action: Report to TOPSPEED.

LOADER_ERROR_ILLEGAL_TRANSFORM (Code 23)

Internal error.

Action: Report to TOPSPEED.

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 TOPSPEED.

Limits and System Requirements

The new executable file format used by overlay model programs 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 increases granularity.

There are also internal limits to the capacity of the loader. The maximum number of modules in a process is 64, and 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 .EXE 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.

The loader uses interrupt 3FH. This must not be used by any other active process.

Multi-thread Programming Using Overlays

By default, the TopSpeed Overlay Manager provides automatic loading and unloading of overlayed segments. However, in a multi-thread program, the restrictions of DOS make it impossible to safely guarantee that a segment is inactive under all conditions. A conservative approach is possible, but this would imply that an overlayed program would not run deterministically, i.e. in memory critical situations the loader could be unable to unload segments, depending on the exact position at which individual threads had been preempted. Thus, multi-thread programs should either manage overlays manually or mark segments used by child threads as resident.

Residency

In a program where the majority of the code executes in the main thread and any child thread performs a simple task, the best approach may be to use residency.

If the thread consists of a very simple procedure, such as a monitor, the best approach may be to mark the segment, and that of any code that it calls, as resident using PRELOAD.

If more complex threads are used a more practical approach is to mark all code as PRELOAD. Only those segments you wish to overlay need then be marked as LOADONCALL. This guarantees that you know exactly which segments are capable of being demand-loaded and automatically unloaded. In this case it must be ensured that only one thread uses the demand-loaded segments.

Manual Segment Unloading

If manual operation is selected using SetMode(LOADER_MANUAL), segments are loaded automatically on demand, but are not unloaded automatically. The procedure UserFlush has no effect. Segments may also be loaded manually. If no memory is available to load a segment, the out-of-memory handler is called .

Segments can be unloaded by a call to UnloadSeg. At the beginning of the main procedure of the program, the call to SetMode must be made. This must happen before the scheduler is started. Segments will be loaded on demand, but will never be unloaded.

When you want to free memory by unloading a segment, a call can be made to UnloadSeg specifying the segment and module. It is up to you to ensure that the segment is inactive in all threads.

Assembly Language Considerations

When programming in any TopSpeed high level language, specifying the overlay model ensures that the correct calling and addressing conventions are used automatically. However, if you are coding sections of your program in assembler you must ensure that the conventions of the overlay model are adhered to. The most important points are:

- All far procedures must create a valid stack frame:

```
push    bp
mov     bp, sp
.........
pop     bp
ret far 0
```

- The BP register must be valid at the point of any far call. The following is therefore illegal:

```
push  bp
mov   bp, sp
mov   bp, 7
call  far MyFunc
pop   bp
ret   far 0
```

- The value of CS is only valid in a non-resident segment for the current call. i.e its value may not be stored in a procedure variable. Procedure variables must be created as 32-bit pointers. The following is acceptable:

```
dd    __myproc
les   di, __myproc
```

but the following is illegal:

```
dw    __myproc
mov   di, __myproc
mov   ax, seg __myproc
mov   es, ax
```

- Code may be reached via a far jump or a far call. A segment that is left by a far jump is then inactive and may unloaded.
- SS, BP and SP must not be changed. The library longjump functions should be used.
- Constants in the current CODE segment may be used. For example:

```
mov ax, cs:[ConstData]
```

- The segment value of a non-preloaded segment may not be stored, since the segment may relocate. However, the value may be passed as a parameter since the segment is guaranteed not to be unloaded while the calling CODE segment is loaded.

CHAPTER 3
DYNAMIC LINK LIBRARIES

Introduction

Dynamic Link Libraries (DLLs) are a major innovation brought about by the introduction of OS/2. The TopSpeed TechKit© allows you to use DLLs with MS-DOS.

DLLs allow your applications to share common data and code which is incorporated into your program at load-time, rather than at link-time, as is the case with traditional linkers. This has two distinct advantages:

- Disc resources are saved, since it is no longer necessary for each executable file to contain its own copy of a function.
- Product updates are much easier, since only specific DLLs need be updated instead of the entire program.

DLLs were introduced into OS/2 for an additional reason — they permit code to be shared at runtime, saving main memory. Under DOS, this is impossible. The ability to create large programs using the TopSpeed Segment-based Overlay System is inherent in using DOS DLLs.

Under TopSpeed, DLLs are created very simply, by using the Project System and a special memory model known as the *dynalink* model. This is a superset of the *multi-thread* model and, under DOS, a superset of the *overlay* model. This means that all the facilities of the TopSpeed Overlay System are available to DLLs and the programs that use them.

The Advantages of Dynamic Linking

For most applications a large .EXE file is acceptable, since segments will be demand-loaded. However, if a project is very large and sections of code are shared, it may be worth making these sections of code DLLs. This reduces the amount of code on disk and speeds up the program make time

DLLs may also be used to distribute product updates. Rather than supply a new version of the whole program, separate DLLs may be updated when necessary.

Pitfalls of Dynamic Linking and Solutions

There are a number of problems associated with using DLLs. However, they can be overcome, provided that the following points are 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. In particular you should master the use of the module definition file (.EXP) described in Appendix B. This is a special file, which is used by the linker to establish the necessary connections between calls to dynamically-linked functions in the executable file, and their code in the DLL.
- 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 exercised 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 may 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.

Rather than using the library DLLs, it is also possible to link one user DLL in a project with the overlay library and export the required functions to other DLLs. The relevant function names must then be listed in the `.EXP` file.

Understanding Dynamic Linking

The best way to appreciate the advantages that DLLs bring is first to examine traditional static linking.

Static Llinking

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. Clearly this is

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.

Even overlays cannot really get around this problem, since they cannot effectively be shared between different applications.

Linking the Traditional Way

Linking is a process by which object modules from both compiled programs and supplied object libraries are joined together 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 be best visualized by 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 program might be:

```
void main( int argc, char *argv[] )
{
  extern int fact( int number );
  printf("Factorial %d is %d\n",3,fact(3));
}
```

These programs could 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 module (see figure .1 below).

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 your 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 in Figure .2, but these 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

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 either:

- Drivers for a wide range of monitors, or
- 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 must adjust every application for the current hardware configuration.

Dynamic Linking

Dynamic linking addresses the problems of 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, regardless of the number of programs 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 works 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: 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.
- Under OS/2 only, it must also 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. Thus, 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; this decision needs to be made on a case by case basis.

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 the module being written.

Writing Dynamic Link Library Modules

There are several points which should be taken into consideration, when writing modules which will be incorporated in a DLL.

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 under OS/2 or the PATH environment variable under DOS.

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 operational demands of the program (Run-time Dynamic Linking).

The DOS DLL facility provided with the TopSpeed TechKit® also includes *Run-time DLL Loading*. See Segment-based overlays API functions, LoadModule, GetProcAddr.

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 module for both static and dynamic libraries under both MS-DOS and OS/2. 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:

- The project must be changed to dos dll (or os2 dll for an OS/2

Dynamic Link Library).

- The model of the project must be dynalink or overlay for DOS, and dynalink or mthread for OS/2.
- 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 Chapter : '*Advanced Library Usage*.' In this case the DLL should use `mthread` (OS/2) or `overlay` (DOS) model.
- You must include the DLL startup 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) must be created:
- For OS/2 all that is required is a list of the symbols to be exported. Under DOS the correct segment attributes must be set. See 'Creating Module Definition Files' below.
- 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. Such references are, in static linking terms, public symbols; in DLL terms they are totally hidden. Only you, as the DLL creator, are aware of their existence and use.

Note: It is not strictly necessary to create a module definition file, if you want to export all public symbols from your DLL. The Project System will create a suitable .LIB file automatically, if no .EXP file is present. See '*Using #implib*' later on in this chapter.

Executing *Make* with a project file which has been constructed in this manner 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* later in this chapter), the DLL needs to be placed so that the run-time loader is able to find it at run-time.

Changing from DOS DLLs to OS/2 DLLs

If you create an MS-DOS DLL and, later, wish to convert it to an OS/2 DLL, the only change required, as long as no operating specific features are used, is to the project file. Simply;

- change dos dll to os2 dll
- run Make.

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 automatically be numbered correctly.

A module definition file for a DOS DLL must also contain the correct SEGMENT statements. The defaults may be copied from overlay.exp:

```
DATA         PRELOAD

SEGMENTS     STACK          PRELOAD
             _INIT          PRELOAD
             CPROC_TEXT     PRELOAD
             PROCESS_TEXT   PRELOAD
             PROC_TEXT      PRELOAD
             PASPROC_TEXT   PRELOAD
             SIG_TEXT       PRELOAD
             EMU_TEXT       PRELOAD
             MP_CODE        PRELOAD
             _DATA          PRELOAD
EXPORTS
  Symbol1       @?
  Symbol2       @?
```

Any preloaded or discardable segments must also be declared in the .EXP file.

A module definition file may be created from an object file by using the utility TSMKEXP. See Chapter : '*Utility Programs*' for further information.

For a complete description of the module definition file syntax see Appendix B.

Multi-thread DLLs

The TopSpeed DLL version of the standard libraries are inherently multi-threaded; you do not need a separate library to make multi-thread DLLs.

However, under DOS, since the Segment-based Overlay System is used by DLLs the restrictions on multi-thread programs apply. See Chapter : '*Segment-based Overlays*' for further details.

Changing DLL Environments

Since all the necessary changes to switch a DLL from MS-DOS to OS/2 are accomplished through the project file, there is no need for you to maintain multiple copies of a given source for different environments.

The project files are small and remain so. They are also easy to update. You would only have to change one source file to maintain a DLL for all environments. The TopSpeed Project System takes care of all the details and interdependencies.

Using #implib

The Project System can produce an import library automatically using the #implib directive. See the "*TopSpeed Developer's Guide*" 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.

Note: An export list may be created very easily by using TSMKEXP, so the use of the module definition file 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 mthread model (OS/2) or overlay model (DOS) if the main program links statically with the standard libraries.

The project file produced using these rules generates an .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 necessary DLLs must be available. The configuration variable LIBPATH is used to specify a path (or paths) to be searched for the appropriate libraries. This is similar to the PATH environment variable used to locate executable programs.

Under MS-DOS, PATH is used. When your program is run, the overlay loader searches for the DLLs it requires in the following order:

1. The current working directory.
2. The "home" directory of the program being executed, if different from the above. The "home" directory of an application is the directory where its `.EXE` file resides, which may not be the current working directory. This search can only be carried out under MS-DOS versions later than 3.00; earlier versions of MS-DOS did not make the "home" directory known to a program.
3. The directories specified in `PATH` in the order given. DLLs are satisfied by the first match found in these places.

If the required DLL cannot be found in any of these places, 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 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), used to create the DLL, looks like this in DOS:

```
#system dos dll
#model dynalink

#compile %main
—  other #compiles as necessary...
#link %prjname
```

The associated module definition file looks like this:

```
DATA         PRELOAD

SEGMENTS     STACK          PRELOAD
             _INIT          PRELOAD
             CPROC_TEXT     PRELOAD
             PROCESS_TEXT   PRELOAD
             PROC_TEXT      PRELOAD
             PASPROC_TEXT   PRELOAD
             SIG_TEXT       PRELOAD
             EMU_TEXT       PRELOAD
             MP_CODE        PRELOAD
             _DATA          PRELOAD
EXPORTS
  mathdemo$fibo    @?  ; Modula-2
  _fibo            @?  ; C
```

The associated project file (MATHDEMO.PR) used to create the DLL, looks like this in OS/2:

```
#system os2 dll
#model dynalink

#compile %main
-  other #compiles as necessary...
#link %prjname
```

The associated moduled definition file looks like this:

```
EXPORTS
  mfibo$fibo       @?  ; Modula-2
  _fibo            @?  ; C
```

When you make the project, two files are produced. MATHDEMO.LIB contains sufficient information to enable the linker to know that the procedures exist. MATHDEMO.DLL contains the executable version of the procedures, to be loaded at run-time.

Using the DLL

The following is a short program containing a reference to fibo:

```
MODULE dlltest;

  IMPORT IO;
  FROM mathdemo IMPORT fibo;

  VAR
    i : CARDINAL;

BEGIN
  FOR i := 0 TO 1000 BY 100 DO
    IO.WrStr("fibo(");
    IO.WrCard(i,1);
    IO.WrStr(") = ");
    IO.WrCard(fibo(i),10);
    IO.WrLn;
  END;

END dlltest.
```

The C equivalent is:

```
                /* DLLTEST.C: */
                /* Demonstrates DLL use */
#include "mathdemo.h" /* Contains prototype */
                /* for fibo() */
#define DEMOVAL (3)
void main( void )
{
  printf("\n** DLL Demonstration Program **\n);
  printf(
    "fibo(%d) = %d\n",
    DEMOVAL,
    fibo(DEMOVAL));
}
```

The associated project file (DLLTEST.PR) could look like this in DOS:

```
#system dos exe
#model dynalink

#compile %main
- other #compiles if necessary
#pragma link(mathdemo.lib)
#link %prjname
```

The associated project file (DLLTEST.PR) could look like this in OS/2:

```
#system os2 exe
#model dynalink

#compile %main
- other #compiles if necessary
#pragma link(mathdemo.lib)
#link %prjname
```

This file uses MATHDEMO.LIB to create DLLTEST.EXE, a program using Dynamic Link Libraries.

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. It is, therefore, helpful to be aware of the procedure.

Before process startup the initialization code defined in initDLL is called. No particular ordering is guaranteed under either DOS or 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 file linked with a DLL.

You must use the default 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  INITCODE
mov     bx, Initrec
mov     ax, INIT_DATA
extrn   __InitLink
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 myinit.a
#pragma linkfirst(myinit.obj)
```

Restriction

No library modules or objects will have been initialized when this initialization code is executed.

Rules and Limitations for DLL Programs

The following restrictions apply whichever source language is used:

- 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.
- Under DOS there is a limit to the number of modules (DLLs) that can be used by a program. This limit is 64 and arises due to limitations on table space. This limit includes the main process and any standard library DLLs.
- The restrictions and programming practices specified for the TopSpeed Segment-based Overlay System must be adhered to for DOS DLLs and programs that use them.

Library Usage and Restrictions

The following restrictions apply:

- Near heap functions are not available when using dynalink models.
- The Modula-2/Pascal SetJmp and Longjmp procedures and LongLabel structure (the setjmp and longjmp functions and the jmpbuf structure in C) 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 DOS see Chapter : '*Segment-based Overlays*'.

For errors under OS/2 see the appropriate Microsoft documentation.

Distributing DLLs

If you intend to distribute your DLLs to a third party, please read the Licence Statement which comes with the distribution disks.

CHAPTER 4
WINDOWS PROGRAMMING

TopSpeed provides a simple method for creating programs and DLLs to run under Microsoft Windows 3.0.

Writing programs for this environment is not simple, and this manual is not designed as a tutorial, or as a reference for Windows 3.0 programming. Before attempting to make your own Windows programs you must be familiar with the Windows API and operating system.

Making Windows Programs

A number of special considerations must be taken into account when creating and making Windows programs:

The Project File

The correct project file options must be set:

- system must be set to win, either by editing the project file or by selecting win from the Project System menu.
- The Project System macro winmath must be set to either emu, chip or none depending on the expected run-time environment. This must be done by editing the project file.
- none - no floating point support included. This option should be selected if your program contains no floating point code.
- emu - floating point emulation. The program will run using a chip if present, or the Windows emulator if not.
- chip - floating point chip support. The program will only run if a chip is present.
- The correct memory model must be selected. Small, compact, medium and large models are supported, although there are restrictions which apply to the use of compact and large models.

The Project System will automatically select and link the correct libraries.

The project file below provides an example of a basic Windows project file, for a program without resources:

```
#system win exe
#model small

#set winmath = "emu"
#compile %main
#link %prjname
```

The Module Definition File

In order to include the correct information in the .EXE file, the linker requires a *module definition file* to be present. This file must have the same name as the executable file, and the extension .EXP.

The first line of the file should be the NAME statement followed by the application type. WINDOWAPI must be selected in the following way:

```
NAME    wdemo   WINDOWAPI
```

A description statement may be included as a comment, although the information will not be included in the .EXE file.For example:

```
;DESCRIPTION 'Sample Application'
```

The stack and near heap size for a Windows program are not taken from the settings of the data(stack_size) and data(heap_size) pragmas; they must be specified in the module definition file using HEAPSIZE and STACKSIZE statements:

```
HEAPSIZE  1024
STACKSIZE 4096
```

A SEGMENTS statement should specify the default attributes for CODE and DATA segments. As a default, CODE segments should be marked PRELOAD MOVEABLE DISCARDABLE. In *Small* and *Medium* models, DATA should be marked PRELOAD MOVEABLE DISCARDABLE. In *Compact* and *Large* models, you must specifiy DATA FIXED. In this case, it will not be possible to to run multiple instances of your program.

```
DATA PRELOAD MOVEABLE MULTIPLE
CODE PRELOAD MOVEABLE DISCARDABLE
```

Any CODE or DATA segments which need different attributes to the defaults specified above, must be specified in a SEGMENTS section. All TopSpeed programs must have the code section _INIT marked as FIXED:

```
SEGMENTS
  _INIT FIXED
```

You can also use this section to mark segments which contain code that is not always used as LOADONCALL. For example, if all code connected with file input/output was in CODE segment FILE_TEXT, you might specify the following:

```
SEGMENTS
  _INIT FIXED
  FILE_TEXT LOADONCALL MOVEABLE DISCARDABLE
```

Finally, any call-back functions must be listed in the EXPORTS section:

```
EXPORTS
  WndProc  @?
  About    @?
```

A complete, basic module definition file for small or medium models looks like this:

```
NAME    wdemo   WINDOWAPI
;DESCRIPTION 'Sample Application'

DATA PRELOAD MOVEABLE MULTIPLE
CODE PRELOAD MOVEABLE DISCARDABLE

SEGMENTS
  _INIT FIXED

HEAPSIZE  1024
STACKSIZE 4096

EXPORTS
  WndProc  @?
  About    @?
```

For a complete description of the module definition file syntax please refer to Appendix B.

Program Source - C and C++

The file windows.h must be included by your source file.

Note: It is important that C++ programmers use the copy of windows.h supplied with TopSpeed to achieve the correct linkage for the Windows API.

The pragma call(windows=>on) must be specified for all functions. This is set automatically by the Project System when #system win is selected. Windows call-back procedures must also use the FAR PASCAL calling convention, and be specified in the EXPORTS section of the .EXP file:

```
long FAR PASCAL WndProc(HWND, unsigned,
                        WORD, LONG);
BOOL FAR PASCAL About(HWND hDlg,
                     unsigned message,
                   WORD wParam,
                     LONG lParam );
```

Program Source - Modula-2

The Modula-2 programmer must create two source files (.MOD) and a definition file (.DEF) for a simple Windows program to ensure that the Modula-2 module initialization mechanism is invoked.

The first source is the main module, which will always have the same format:

```
(*# data(stack_size => 0) *)
MODULE MWdemo;

IMPORT MWMain, Windows;

BEGIN
  Windows.EntryPoint;
END MWdemo.
```

In the main module the definition for the Windows API must be imported, with the definition file for the program implementation. The call to Windows.EntryPoint will invoke the Windows initialization which in turn calls WinMain, defined in the program implementation.

Since Windows supplies a stack segment, the program stack size must be set to zero.

The second source file is the actual implementation of the program. All the Windows functions and the function WinMain must appear in the corresponding definition module (.DEF file):

```
DEFINITION MODULE MWMain;

IMPORT Windows;

(*# name(prefix=>windows) *)
(*# call(near_call=> off,            reg_param=>()) *)

PROCEDURE About (hDlg: Windows.HWND;
                 message: CARDINAL;
                 wParam : WORD;
                 lParam : LONGINT
          ) : Windows.BOOL ;

PROCEDURE WndProc(hWindow: Windows.HWND;
            message: CARDINAL;
            wParam: CARDINAL;
            lParam: LONGINT
          ): LONGINT ;

PROCEDURE WinMain(hInstance: Windows.HANDLE;
            hPrevInstance: Windows.HANDLE;
            lpszCmdline: Windows.LPSTR;
            cmdShow : INTEGER
          ) : Windows.BOOL;
END MWMain.
```

The pragma settings in the definition module for the program implementation are important:

- The Windows call-back functions and WinMain are far call, stack parameter calling convention.
- The naming convention for the Windows call-back functions and WinMain is prefix=>windows.

- Any functions which are not call-back functions (except WinMain) should have normal JPI calling conventions, with the exception that pragma call(windows => on) is in force. The Project System sets this pragma automatically when #system win is specified.

Note: All the Windows call-back functions (except WinMain) in the definition file must be exported in the module definition file (.EXP).

Program Source - Pascal

The Pascal programmer must create two source files (.PAS) and an interface file (.ITF) for a simple Windows program to ensure that the Pascal module initialization mechanism is invoked.

The first source is the program, which will always have the same format:

```
(*# data(stack_size => 0) *)
program PWdemo;

IMPORT PWMain, Windows;

begin
  Windows.EntryPoint;
end.
```

In the program unit the interface file for the Windows API must be imported, together with the interface file for the actual program implementation. The call to Windows.EntryPoint will invoke the Windows initialization which in turn calls WinMain, defined in the program implementation.

Since Windows supplies a stack segment, the program stack size must be set to zero.

The second source file is the actual implementation of the program. All the Windows call-back functions and the function WinMain must appear in the program interface file (.ITF).

```
INTERFACE UNIT PWMain;
IMPORT Windows;
(*# name(prefix=>windows) *)
(*# call(near_call=> off,            reg_param=>()) *)

function About (hDlg: Windows.HWND;
            message: word;
            wParam : word;
            lParam : LONGINT    ) : Windows.BOOL ;

function WndProc(hWindow: Windows.HWND;
            message: word;
            wParam:  word;
            lParam: integer
          ): integer;

function WinMain(hInstance: Windows.HANDLE;
            hPrevInstance: Windows.HANDLE;
            lpszCmdline: Windows.LPSTR;
            cmdShow : int16) : Windows.BOOL;
end.
```

The pragma settings in the interface unit for the program implementation are important:

- The Windows call-back functions and WinMain are far call, standard parameter calling convention.
- The naming convention for the Windows call-back functions and WinMain is prefix=>windows.
- Any functions which are not call-back functions (except WinMain), should have normal JPI calling conventions, with the exception that pragma call(windows => on) is in force. The Project System sets this pragma automatically when #system win is specified.

Note: All Windows call-back functions (except WinMain) in the interface unit must be exported in the module definition file (.EXP).

Memory Management

The segment attributes of static data and code can be controlled with SEGMENTS statements in the .EXP file. For more details please refer to Appendix B and the *Microsoft Windows Software Development Kit* documentation.

Dynamic Memory

All library functions concerned with memory allocation are mapped to Windows *NearHeap* allocation functions. It is recommended that Windows' memory allocation functions are used directly, rather than the standard library functions, since space in the near heap is limited, and greater control can be gained by using the Windows API directly.

Debugging

TopSpeed's debugger, VID, does not support debugging under Windows. However, TopSpeed includes the utility VID2CV which converts VID debug information to CodeView format.

Two operations are required to produce an executable file that can be debugged under Windows:

- Full debug information must be generated for the program either by selecting full from the Project Vid menu, or by editing the project file and inserting the debug(vid=>full) pragma.
- VID2CV must be run after the program has been linked, using the following command line:

```
VID2CV progname
```

The Resource Compiler

The TopSpeed TechKit© includes the Resource Compiler TSRC.EXE, which, in conjunction with the TopSpeed Linker, allows resources to be added to executable programs for use under Microsoft Windows Version 3, without requiring the use of the Microsoft Windows Software Development Kit. The TopSpeed Resource Compiler does not work in the same way as the Microsoft resource compiler, RC.EXE, but it does make use of the same file formats.

The TopSpeed Resource Compiler processes a *resource script file* (with a .RC extension) and produces a compiled resource file (with a .RES extension) of the same name. This file can then be added to the link list (for example using #pragma link(myfile.res)), and the TopSpeed Linker will automatically add the resources to the end of the executable file.

The format for resource information is based on C syntax, so the header file WINSTYLE.H is provided to supply commonly used definitions for programmers in all TopSpeed languages.

The configuration file TSPRJ.TXT contains a #declare_compiler command for the file extension .RC, so that a resource file can be added to a project simply by including the statement:

```
#compile resfile.rc
```

in the project file. The TopSpeed Project System will then build the .RES file, if necessary, and then add it to the link list.

A Windows Program - the Complete Sequence

The steps required to make a Windows executable file are as follows:

1. Create the following files:

source files
resource (.RC) file
module definition file

2. Within the project file, specify `#system win exe`, and a suitable `#model` command.
3. Compile the source files and resource script file using the `#compile` project command.
4. Create the executable file with a `#link` command. This automatically includes the program's resources in the `.EXE` file.
5. If you intend to debug the program with CodeView, `VID2CV` should be executed. This can be done within the project file using a conditional statement as in the example below:

```
#system win exe
#model small
#pragma debug(vid=>full)

#set winmath = "none"
#compile cwdemo.c
#compile cwdemo.rc
#link %prjname

#if "%make"="on" #and #not "%action"="compile"
  #and #not debug(vid)=off #then
    #run "vid2cv %prjname%.exe"
#endif
```

Modula-2 Library Extensions

Two modules are available to Modula-2 programmers for handling far data in *small* and *medium models*. These modules are intended for handling strings loaded from resources or provided by Windows. Such strings are not in the default data segment, and so cannot be handled by the normal FIO and Str modules in these models.

In other memory models, the standard library modules operate on far data, and so can be used in all cases.

WinFio

```
PROCEDURE Open (
  Name: ARRAY OF CHAR) : File;
PROCEDURE OpenRead(
  Name: ARRAY OF CHAR) : File;
PROCEDURE Create (
  Name: ARRAY OF CHAR) : File;
PROCEDURE Close (F: File);

PROCEDURE Exists(
  Name: ARRAY OF CHAR) : BOOLEAN;
```

```
PROCEDURE WrBin (
  F: File;
  Buf: ARRAY OF BYTE;
  Count: CARDINAL);
```

```
PROCEDURE RdBin (
    F: File;
    VAR Buf: ARRAY OF BYTE;
    Count: CARDINAL) : CARDINAL;
```

WinStr

```
PROCEDURE Compare(
  S1,S2:ARRAY OF CHAR):INTEGER;
PROCEDURE Length(
  S:ARRAY OF CHAR):CARDINAL;
PROCEDURE Concat(
  VAR R:ARRAY OF CHAR;
  S1,S2:ARRAY OF CHAR);
PROCEDURE Append(
  VAR R:ARRAY OF CHAR;S:ARRAY OF CHAR);
PROCEDURE Copy(
  VAR R:ARRAY OF CHAR;S:ARRAY OF CHAR);
PROCEDURE Delete(
  VAR S:ARRAY OF CHAR;P,L:CARDINAL);
PROCEDURE Insert(
  VAR S1:ARRAY OF CHAR;
  S2:ARRAY OF CHAR;P:CARDINAL);
PROCEDURE CharPos(
  S:ARRAY OF CHAR;C:CHAR):CARDINAL;
PROCEDURE Caps(
  VAR S:ARRAY OF CHAR);
PROCEDURE Lows(
  VAR S:ARRAY OF CHAR);
PROCEDURE Slice(
  VAR R:ARRAY OF CHAR;
  S:ARRAY OF CHAR;P,L:CARDINAL);
PROCEDURE Pos(
  S,P:ARRAY OF CHAR):CARDINAL;
PROCEDURE NextPos(
  S,P:ARRAY OF CHAR;
  Place:CARDINAL):CARDINAL;
PROCEDURE RCharPos(
  S:ARRAY OF CHAR;C:CHAR):CARDINAL;
PROCEDURE Item(
  VAR R:ARRAY OF CHAR;S:ARRAY OF CHAR;
  T:CHARSET;N:CARDINAL);
PROCEDURE ItemS(
  VAR R:ARRAY OF CHAR;S,T:ARRAY OF  CHAR;N:CARDINAL);
PROCEDURE Prepend(
  VAR S1:ARRAY OF CHAR;S2:ARRAY OF CHAR);
PROCEDURE Subst(
  VAR S1:ARRAY OF CHAR;
  Target,New:ARRAY OF CHAR);
PROCEDURE Match(
  Source,Pattern:ARRAY OF CHAR):BOOLEAN;
PROCEDURE FindSubStr(     Source,Pattern:ARRAY OF CHAR;
  VAR pos:ARRAY OF PosLen):BOOLEAN;
```

```
PROCEDURE IntToStr(
  V:LONGINT;VAR S:ARRAY OF CHAR;
  Base:CARDINAL;VAR OK:BOOLEAN);
PROCEDURE CardToStr(
  V:LONGCARD;VAR S:ARRAY OF CHAR
  Base:CARDINAL;VAR OK:BOOLEAN);
PROCEDURE StrToInt(
  S:ARRAY OF CHAR;Base:CARDINAL;
  VAR OK:BOOLEAN):LONGINT;
PROCEDURE StrToCard(
  S:ARRAY OF CHAR;Base:CARDINAL;
  VAR OK:BOOLEAN):LONGCARD;
PROCEDURE StrToC(
  S:ARRAY OF CHAR;VAR D:ARRAY OF CHAR):BOOLEAN;
PROCEDURE StrToPas(
  S:ARRAY OF CHAR;
  VAR D:ARRAY OF CHAR): BOOLEAN;
```

Library Limitations

There are a number of library limitations which must be considered when programming for Windows:

C and C++ Language

The following functions may NOT be used:

- All text windowing and console I/O functions. (window.h and conio.h).
- All file I/O functions that use the predefined streams. In general, file buffering is not recommended since files should not be left open for extended periods.
- All process and multi-thread functions (process.h), and delay.
- All bios and dos interface functions not prototyped in bios.h and dos.h when macro _WINDOWS is defined.
- All memory allocation functions except malloc, calloc and free.
- All mouse interface functions, (mouse.h).
- All graphics functions in graph.h.

Modula-2 Language

The following modules are NOT included in the library when producing a Windows program:

```
Lim, IO, Graph, MsMouse, Process,
Window, BiosIO.
```

In addition, the following functions/procedures may NOT be used:

- All file I/O functions that use the predefined streams. In general,

file buffering is not recommended since files should not be left open for extended periods.

- Any process control functions in Lib such as Exec, ExecCmd etc.
- Lib.Delay.

Pascal Language

The following modules are NOT included in the library when producing a Windows program:

```
PasWin, PasProc, Crt, BGI.
```

In addition, the following functions/procedures may NOT be used:

- All file I/O functions that use the predefined files. In general, file buffering is not recommended since files should not be left open for extended periods.
- Any process control functions in PasDos such as Exec.

Any attempt to use an unsupported feature may cause link or run-time errors.

Using Windows Version 2

The resource compiler from the Windows 2 Software Development Kit must be used when making programs for Windows 2.

Making Windows DLLs

Before attempting to use DLLs under Windows you must be familiar with the relevant documentation in the *Windows Software Development Kit* and the information provided in Chapter 3: '*Dynamic Link Libraries*'.

The Project File

The DLL project file should select Windows operating system and DLL filetype:

```
#system win dll
```

Any legal Windows memory model may be selected:

```
#model small | medium | compact | large
```

The macro winmath must be selected depending on the type of floating point support required. For example:

```
#set winmath = "none"
```

The Project System will automatically select and link with the correct libraries.

DLL Initialization and Termination

The Project System automatically includes the Windows DLL initialization file initwDLL. This calls the DLL library main function LIBMAIN which should be defined in a separate source file and included in the project:

```
#compile libmain
```

The libmain main function initializes the library on its first invocation:

```
#include “windows.h”

int PASCAL LIBMAIN(
  HANDLE hInstance,
  WORD wDataSeg,
  WORD cbHeapSize,
  LPSTR lpszCmdLine)
{
  int ret = 1;
  if (!hInstance) {
  /* Call initialization */
            /* procedure if this  */
  /* is first instance */
  if(!LocalInit(wDataSeg, 0,
                cbHeapSize) == 0)
  ret = 0;                   /* error */
  }
  return ret;
}
The termination function WEP must also be defined.
short FAR PASCAL WEP(int nParam)
{
    if(nParam == WEP_SYSTEM_EXIT) {
      return 1; /* system exit */
    }
  if(nParam == WEP_FREE_DLL) {
  return 1; /* use count is zero */
               /* - DLL freed */
    }
    else
  return 1;       /* undefined, just return */
}
```

Although only a C example is given here, full examples of library initialization and termination functions are provided for each language in the following modules:

C/C++	LIBMAIN.C
Modula-2	LIBMAIN.MOD, LIBMAIN.DEF

Pascal LIBMAIN.PAS, LIBMAIN.ITF

Module Definition File

There are two extra requirements for a Windows DLL module definition file:

- The WEP termination function must be exported as well as all functions or procedures forming the interface.
- DATA must be set to single.

Entry points to a DLL must be declared using the Windows calling convention.

The following example is a module definition file for a DLL exporting the function MessagePaint:

```
LIBRARY    wdllc
;DESCRIPTION 'Sample DLL'
DATA    PRELOAD SINGLE
HEAPSIZE  1024
STACKSIZE 4096
EXETYPE WINDOWS
EXPORTS
    MessagePaint  @?
    WEP           @?
```

A Complete DLL Project File

```
#system win dll
#model small jpi
#pragma debug(vid=>full)

#set winmath = "none"
#compile libmain.c
#compile wdllc.c
#link %prjname%.dll
#if "%make"="on" #and #not "%action"="compile"
#then
  #run "vid2cv %prjname%.dll"
#endif
```

Although a Windows DLL may use any legal Windows memory model, the interface must use far data and calls.

CHAPTER 5
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.
- Interfacing to assembly language.

This chapter cannot be a tutorial for all TopSpeed languages, thus a knowledge of the languages concerned is assumed.

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. Inclusion of these files automatically causes the correct libraries to be linked when you make your program.

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 may then be used in the same way as a regular C file.

Creating Your Own Cross Definition Files

There are three main tasks involved in the creation of an efficient inter-language interface:

- The translation of the appropriate declarations.
- The setting up the of the necessary calling conventions.
- The setting up 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.

Type Equivalents

The following table lists the type equivalences between the available TopSpeed language products:

C & C++	Modula-2	Pascal
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. However, in Modula-2 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, which is 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 used, and the pragma call (t_l_size=>on) specified:

```
(*# call(t_l_size=>on) *)
function Length(var s): word;
```

When passing a translated Pascal string to C, the typeless var parameter is used with the pragma call(t_l_size=>off):

```
(*# call(t_l_size=>off) *)
function strlen(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 each parameter to be passed explicitly and the pragma call(o_a_size=>off) to be specified:

```
(*# 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. C and C++ use a 16-bit type, while Modula-2 and Pascal use an 8-bit type.

The call data(var_enum_size=>on) pragma must be used in Modula-2 and Pascal to achieve compatibility.

Calling Conventions

In addition to setting up the string calling conventions mentioned above, the overall calling convention must be specified, whenever multi-language programming is employed.

The basic JPI calling convention is consistent between all languages. However, 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 will 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 will define the Modula-2 calling convention for any TopSpeed C or C++ program:

```
#pragma call(c_conv=>off)
```

Pascal to C

The following pragma will define 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 will define the Pascal calling convention for any TopSpeed C or C++ program:

```
#pragma call(c_conv=>off)
```

Pascal to Modula-2

The following pragma will define 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 will define 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 will define 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 will define the Modula-2 and Pascal naming convention for any program written using TopSpeed C or C++:

```
#pragma name(prefix=>"")
unsigned Str$Length(unsigned size,
                    char *);
unsigned MyModule@CardinalVar;
```

As names in TopSpeed Modula-2 and TopSpeed Pascal are overloaded between modules, 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;
```

When using TopSpeed C++ the correct linkage specifier must be used:

```
extern "Modula2"
extern "Pascal"
```

C++ linkage specifiers for Modula-2 and Pascal may also contain a module name.

Pascal to Modula-2

The following pragma will define the Modula-2 calling convention for TopSpeed Pascal programs:

```
(*# call(t_l_size=>on) *)
(*# module(implementation=>off) *)
```

Modula-2 to Pascal

No action is necessary, since the naming conventions are identical apart from case restrictions.

Library Considerations

There are a number of important considerations which must be taken into account 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
"*Modula-2 Library Reference*".

Care should be taken that streams or files have the same access and buffering modes.

The Pascal I/O model is different to that of C and Modula-2 and care should 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 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 - 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 or static destructors are 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, apart from in the case of an exception such as an 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. which do not specify module(init_code=>off):

```
#pragma save
#pragma name(prefix=>"")
          /* Must not have _ prefix */
extern char MyModule$[];
#pragma restore
```

Assembly Language Interface

Interfacing to assembly language presents its own, special group of problems and needs:

Standard C

The standard C parameter passing mechanism is summarized as follows:

- Each parameter is pushed on to the stack, starting at the rightmost parameter.
- The function is called.
- Upon return from the function, the stack is restored to the state it was before the function call began by the caller.

This mechanism allows C to call the same function with differing numbers of parameters, for example:

```
printf("%d squared = %d\n",i,i*i);
printf("%d cubed = %d; squared = %d\n",i,i*i*i,i*i);
```

C can do this because the *calling* program knows how many parameters were pushed onto the stack and can, therefore, tidy it up afterwards. The function itself does not have to worry about cleaning up the stack, as long as it has some way of knowing how many parameters to expect. printf() knows how many parameters to expect from the format string.

The three main differences in the JPI calling convention are:

- Parameters are passed, as far as possible, in the CPU's registers. The stack is only used when the registers are used up.
- Parameters are passed left to right. This is the opposite direction to standard C.
- The called function restores the stack if parameters have been pushed.

Standard Pascal and Modula-2

The standard Pascal/Modula-2 parameter passing mechanism is summarized as follows:

- Each parameter is pushed on to the stack, starting at the leftmost parameter.
- The procedure is called.
- Upon return from the function, the stack is restored to the state it was before the function call began by the called procedure.

The main differences in the JPI calling convention are:

- Parameters are passed, as far as possible, in the CPU's registers. The stack is only used when the registers are used up.

The JPI Calling Convention

The JPI calling convention comprises:

Simple Types

- Regardless of the normal calling conventions of the language, the parameters are processed from left to right and assigned to registers. 8-bit and 16-bit values are passed in one register. 32-bit values are passed in register pairs.
- The values are assigned to the 80x86 registers in the following order:

  `ax` then `bx` then `cx` then `dx`
- When all the default registers have been assigned, the rest of the parameters (if any) in the same order, left to right, are pushed onto the stack. It cannot be expressed too strongly that this is the opposite order to standard C. This means that modules compiled using other vendor's compilers will not work with functions compiled under the JPI calling convention. See 'Interfacing to Third Party Code' in the "TopSpeed Developer's Guide".
- If parameters were passed on the stack, the called function pops the required number of bytes to restore the stack to its original state.

The following points should be noted:

- 32-bit values are never split between registers and the stack. If no register pair remains, the entire value is pushed on the stack.
- If a register remains unused and a parameter will fit, even if the stack has already been used, the register will be used by that parameter.
- Floating point values are passed in the following registers: st0, st6, st5, st4, st3. Note that they are stored in the registers, not pushed onto the coprocessor stack.
- Functions declared with variable parameter lists (e.g, void func(char * f, ...)) will always use the stack for parameter passing. This underlines the importance of using function prototypes to ensure that parameter type checking is enforced. If TopSpeed C doesn't know the correct prototype for a function, the generated calling sequence will be wrong.

- Structures are always passed on the stack.

Array and String Types

```
C and C++
```

In C and C++ all arrays are passed as an address. Therefore, they will follow the convention for 16- or 32-bit simple types, depending on the memory model.

Modula-2

In Modula-2, open arrays are handled internally as two parameters. The array size is passed first, as a 16-bit value, followed by the address of the array, as either a 16- or 32-bit value, depending on memory model. An open array passed by value will be copied by the called procedure.

This convention can be modified by pragmas:

- The call(o_a_copy=>off) pragma prevents the called procedure making a local copy of the array.
- The call(o_a_size=>off) pragma suppresses the passing of the array size word.
- The call(ds_eq_ss) pragma overrides the default pointer size used for the address of the array.

Pascal

In Pascal, dynamic strings are handled internally as two parameters. First the array size is passed as a 8-bit value, followed by the address of the array, as either a 16- or 32-bit value, depending on memory model. A string passed by value will be copied by the called procedure.

Conformant arrays are handled internally as four parameters. First, the array size, low bound and high bound are passed as 16-bit values (if int16 is used as the type), in that order. Then the address of the array, as either a 16- or 32-bit value, depending on memory model. A conformant array passed by value will be copied by the called procedure.

This convention can be modified by pragmas:

- The call(ds_eq_ss) pragma over-rides the default pointer size used for the address of the array.

All structured type

value parameters are passed on the stack, with the exception of C++ classes for which a copy constructor is defined.

Variable Argument List Functions

If the call(opt_var_arg) pragma is set on (the default), a special calling convention is used to reduce code size. If this pragma is set off, normal stack frame parameter passing is used.

The optimized calling convention uses a table of near calls to the actual function, with a stack pop to allow the function to clear its own stack. The following example illustrates the technique when defining the function in assembly language.

Note: The example given is for a function that is *far* called - for a near called function, the ret instructions would all be near, and the value of arg1 is 6. The calls to ActualFunction are always *near*.

```
public _VarArgFunc :

    call ActualFunction
    ret  far 0
    call ActualFunction
    ret  far 2
    call ActualFunction
    ret  far 4
    call ActualFunction
    ret  far 6
    call ActualFunction
    ret  far 8
    call ActualFunction
    ret  far 0AH

arg1    =  8

ActualFunction:

    push bp
    mov  bp, sp
    sub  sp, locals
    push ax
    mov  ax, [bp][arg1]
    (* .... *)
    pop  ax
    mov  sp, bp
    pop  bp
    ret  near 0
```

When calling an optimized variable argument list function from assembly language, the correct offset from the public label must be used depending on the number of words passed as parameters:

1 word - label + 04H.
2 words - label + 0AH.
3 words - label + 10H.
4 words - label + 16H.
5 words - label + 1CH.
6 words, or more - call label. (The caller must pop the stack in this case).For example,

```
mov  ax, 1
push ax
mov  ax, _format
push ax
call _printf+0AH

mov  ax, 1
push ax
push ax
push ax
push ax
push ax
mov  ax, _format
push ax
call _printf
add  sp, 0CH
```

Typeless Parameters

In Pascal, typeless parameters are handled internally as two parameters. The size of the object is passed first as a 16-bit value, then the address of the object is passed as a 16- or 32-bit value depending on the memory model. If the parameter is passed by value, the called function will make a local copy depending on the setting of the call(t_l_copy) pragma.

This convention can be modified by pragmas:

- The call(t_l_size=>on) pragma causes a local copy of a typeless value parameter to be made.
- The call(t_l_size=>off) pragma suppresses the passing of the size word.
- The call(ds_eq_ss) pragma over-rides the default pointer size used for the address of the object.

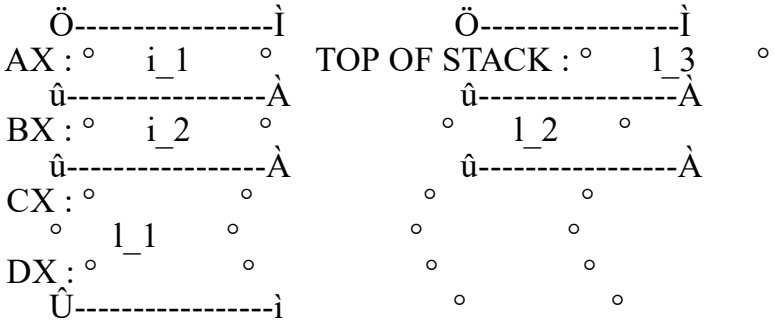
Examples of the JPI Calling Convention

The examples below illustrate the way that the JPI calling convention works:

A call of the function:

```
rover( int i_1, int i_2, long l_1, long l_2, long l_3 );
```

results in the following register and stack configuration:

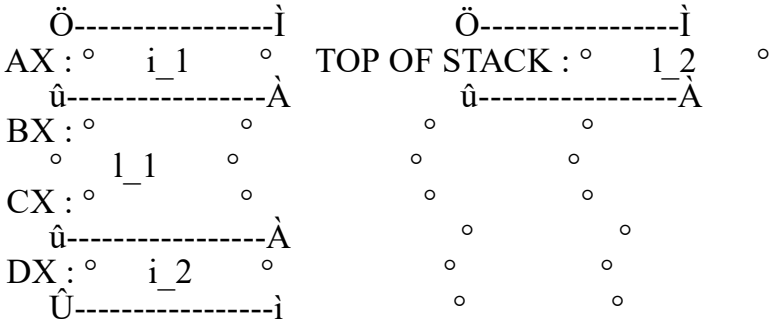


Note: l_3 is above l_2 on the stack because l_2 is pushed onto the stack first. This example shows the left to right processing and the allocation of registers.

The next example shows how TopSpeed generates the calling sequence, when a register remains after the stack has already been used. The function call:

```
rover( int i_1, long l_1, long l_2, int i_2 );
```

results in the following register and stack configuration:



Note: i_2 is placed in DX and l_2 on the stack. When the compiler processes l_2 there is no register pair remaining for the long, 32-bit, value l_2, therefore it is pushed on the stack. However, when the compiler reaches i_2 (a 16-bit value), it puts this value in DX because this has not already been used.

Thus, the JPI calling convention makes maximum use of the registers of the 80*x*86 CPU, when passing parameters to a function.

Returning Values to TopSpeed High-level Functions

Values are returned in the 80*x*86 registers. The general convention is:

- 8-bit and 16-bit values are returned in ax.
- 32-bit values are returned in dx:ax.

Return Registers for C Objects:

- char and int values are returned in ax.
- long int values are returned in dx:ax.
- near pointer values are returned in ax.
- far pointer values are returned in dx:ax.

Return Registers for Modula-2 Objects:

- CHAR, SHORTCARD, INTEGER and CARDINAL values are returned in ax.
- LONGINT and LONGCARD values are returned in dx:ax.
- NearADDRESS values are returned in ax.
- FarADDRESS values are returned in dx:ax.

Return Registers for Pascal Objects:

- char, byte, int16 and word values are returned in ax.
- integer values are returned in dx:ax.
- nearaddress values are returned in ax.
- faraddress values are returned in dx:ax.

Functions Returning Floating-point Values

JPI Calling Convention

Floating-point values are returned in the top register of the 80*x*87 stack. The following code fragment illustrates the way floating-point values are returned:

```
mov  bx, sp
fld  qword st(0), ss:[bx][fabs_x]
                  (* Get the parameter *)
fabs st(0)         (* Result in st(0) *)
fstp st(1), st(0) (* Pop and leave result *)
                  (* in st(0) *)
ret  0
```

Standard Stack Model

A pointer to a static memory location is returned. This contains the return value.

Functions Returning Structures

Functions that return structures are passed, as an implicit first parameter, a 16-bit pointer to a temporary buffer in the stack. The functions declared parameters follow this.

The called function copies its return value into this buffer. On return, the caller then copies that return value from the temporary buffer to its final destination.

Register Preservation

JPI Calling Conventions

Normally, TopSpeed functions explicitly preserve the contents of the following registers during a function call:

```
ax bx cx dx
ds di si bp
st1 st2
```

In addition, the ds register is *always* preserved as it is the pointer to the current function's DATA segment. The bp register is also implicitly preserved as the base pointer for the current function's local variable work space. All functions automatically save bp on the stack on entry, and restore it before they return.

Note: The general purpose registers ax, bx, cx and dx are only preserved by a function call when they are used neither for passing parameters nor for returning results.

Standard Stack Model

Only the standard set of registers are preserved:

```
ds si di bp
```

C Argument Type Promotions

C automatically *promotes* the type of function arguments in the absence of other information. For example, if the following example is used without a prototype for spot():

```
void main( void )
{
  float x = 1.7;
  char y = 'D';
  int i;
  i = spot(x,y);
}
```

then x will be promoted to a double and y to an int before the function is called. This is discussed more fully in the "*TopSpeed C Language Reference*".

This is a potential source of problems if proper prototyping is not used. In the above example, spot() would produce erroneous results if it only expected a float and char as parameters.

Linkage Names

C Language

Unless over-ridden by a name pragma, an underbar is prepended to the identifier:

```
myfunc becomes _myfunc.
```

C++ Language

A complex name scrambling algorithm is used to produce unique names. See the "*TopSpeed C++ Language Reference*" for further details.

Modula-2

Unless over-ridden by a name pragma, the module name followed by a separator is prepended to the identifier. If the identifier refers to a procedure the separator is $, while if it refers to data the separator is @:

```
DEFINITION MODULE MyMod;

PROCEDURE MyProc(); becomes MyMod$MyProc

VAR
  MyVar: CARDINAL; becomes MyMod@MyVar
```

Pascal

Unless over-ridden by a name pragma, the module name followed by a separator is prepended to the identifier. The name is also forced to upper case. If the identifier refers to a procedure the separator is $. If the identifier refers to data the separator is @:

```
Interface unit MyUnit;

procedure MyProc; becomes MYUNIT$MYPROC;

var
  MyVar: word; becomes MYUNIT@MYVAR;
```

C and C++ Low Level Initialization

The following mechanism is used by the C library and static C++ objects. The following example illustrates the technique:

```
include “corelib.inc”

section

segment _INIT(INIT_CODE,28H)
segment _TEXT(CODE,28H)
segment _DATA(DATA,28H)
segment _BSS(BSS,28H)

select _TEXT
do_myinit :    (* refers to code below *)

select _INIT   (* magic segment *)
(*%F NearCall *)
initio : dw MAGIC_NUM
         db 2  (* the priority *)
         dd do_myinit
(*%E *)
(*%T NearCall *)
initio : dw MAGIC_NUM
         db 2  (* the priority *)
         dw do_myinit
(*%E *)
select _DATA
public __key_variable :  dw  0, initio

select  _TEXT
               (* do_myinit refers to here *)
    push bp

    .... your code
    extrn __InitLoop
(*%F NearCall *)
    call far __InitLoop
    pop  bp
    ret  far 0
(*%E *)
(*%T NearCall *)
    call __InitLoop
    pop  bp
    ret  0
(*%E *)
```

The following points must be noted:

- The segmentation must be followed precisely.
- The priority byte may be altered as follows.
- The default priority is 16. See module(init_prio) pragma in the “TopSpeed Developer’s Guide”.
- Higher priorities are executed first
- Priorities from 20 upwards are reserved for low-level standard library initialization.
- The C++ library has priority 22.
- The variable __key_variable must be referenced for the init code to be linked.

- The actual user code must preserve all registers corrupted.
- The call to __InitLoop must be executed.
- MAGIC_NUM and other symbols are defined in corelib.inc.

Modula-2 and Pascal Initialization

Modula-2 and Pascal use the same mechanism for initializing static objects and modules. Dependencies are calculated at run-time, including any initialization code in imported DLLs. The following rules apply:

- The order of initialization mutually referential modules is undefined.
- At the end of the chain of the initialization code, the main module is called.
- Initialization code may be written in assembly language.

A variable length structure is created comprising first, the address of the actual initialization code, followed by a list of imported module init structures, and terminated by a word value -1:

```
segment _TEXT(CODE,28H)
select _TEXT
$init:          (* The init code *)
  ... user initialization code
(*%T NearCall*)
ret near 0
(*%E*)
(*%F NearCall*)
ret far 0
(*%E*)

segment _DATA(DATA,00028H)  (* the record *)
extrn Str$
extrn Lib$
public Spawn$:
(*%T NearCall*)
dw $init
(*%E*)
(*%F NearCall*)
dd $init
(*%E*)
(*%F SameDS *)
dd Str$
dd Lib$
(*%E *)
(*%T SameDS *)
dw Str$
dw Lib$
```

```
(*%E *)
dw -1
```

Reserved Segments and Groups

The following segmentation restrictions apply:

- The group DGROUP is reserved and its ordering must not be altered.
- The following segments are reserved:

```
ENTERCODE (CODE,28H)
INITCODE  (CODE,28H)
SIG_TEXT  (SIG_CODE,68H)
EMU_TEXT  (EMU_CODE,28H)
PROC_TEXT (CODE,68H)
MP_CODE   (MP_CODE,68H)
_INIT  (INIT_CODE,28H)
STACK  (STACK,74H)
```

Floating Point Code Generation

When using the JPI calling convention, a special mode of floating point register usage is employed, in which the 80*x*87 stack operates empty. One push is allowed to store a register, which is then loaded in to the desired destination register, emptying the stack.

You should study the disassemblies and the library module CoreMath.a. However, you may prefer to write code using the usual stack based technique. In this case, stack model libraries will be required.

CHAPTER 6
TOPSPEED ASSEMBLER

The TopSpeed Assembler is a single-pass assembler used to assemble source files into .OBJ form. The resulting object files can be linked with programs written in other TopSpeed languages. The TopSpeed Project System automatically detects that a particular module is written in assembly language and invokes the TopSpeed Assembler as required. Assembly language files are identified by a .A extension on the filename. Thus, INITDLL.A is the assembly language source code for one part of the TopSpeed library.

Overview

The TopSpeed Assembler is designed to be simple to use and fast to operate. The TopSpeed Assembler supports smart linking and interfaces to the other languages in the TopSpeed family. The TopSpeed assembly language differs from "standard" 8086 assemblers in a number of ways:

- The lexical structure is derived from Modula-2. In particular, the Assembler is case sensitive, comments are marked with (* and *) and hexadecimal constants must be in upper case.
- A semi-colon (;) is used to separate multiple statements on a line. In fact, a newline and a semi-colon are regarded as identical by the Assembler.
- There is only one kind of label and there are no data types. Instead of data types, TopSpeed Assembler uses specifiers. Thus, the specifiers near, far, byte and word can be used to define the data type of the operands of an instruction.
- Memory operands and segment overrides must always be explicitly specified.
- No macros or other complicated features are used.

This chapter describes the assembly language which is recognized by the TopSpeed Assembler. This chapter does *not* contain a detailed discussion of any assembly-level programming. The 8086 and 8087 instruction sets are described in Appendix D.

The Assembler achieves its high speed by assembling the source file in a single pass. This means that variables and labels must be defined before they are used, otherwise the Assembler assumes that the item is to be defined later in the same segment. If the item is not found there, an error message is produced.

Invoking the Assembler

The TopSpeed Project System automatically invokes the Assembler for any source files in a project with a .A extension. This means that you do not have to worry about the interdependencies of a project, or how those interdependencies are maintained.

The Assembler can be invoked from within the TopSpeed Environment by selecting Compile from the main menu when a .A file is loaded (or by using the short-cut, *Alt-C*). Assembly errors are reported in the same way as compiler errors: the error window is displayed and you can enter corrections.

Program Layout

A TopSpeed Assembler program has the following general layout:

```
module <name>(* Names the module *)

segment <name>.....(* Define segments *)

<code and data>....

section       (* Another section - Optional *)
....<and so on>

end           (* Marks the end of the module *)
```

Segments

An 8086 program consists of one or more segments. Each segment contains either code or data. Every program contains at least one segment: a CODE segment consisting of the executable code of the program.

Segments are declared using the following syntax:

```
segment <segment-name> ( <class> , <alignment> )
```

For example, the statement:

```
segment SPOT (CODE, 28H)
```

declares a segment called SPOT of the class CODE, with an alignment code of 28H. The alignment code determines the type of address the segment is to be located at when the program is linked.

The normal values are:

8H	a byte aligned segment. The segment can be loaded at any address.
48H	a word aligned segment. The segment must be loaded at an even address, with a gap of one byte being left if necessary.

68H — a paragraph aligned segment. The segment must be loaded at an address which is exactly divisible by 16. A gap of up to 15 bytes is left if necessary.

74H — a stack segment. This is only used for segments which are to be used as a stack.

Segments can also be grouped together. Groups must be declared after the segments they contain, using the following syntax:

```
group <group-name> ( <segment-name-list> )
```

For example:

```
group G_FOO (A_FOO, B_FOO, C_FOO)
```

declares a group called G_FOO consisting of the three segments A_FOO, B_FOO and C_FOO. This must be preceded by the definitions of A_FOO, B_FOO and C_FOO.

The TopSpeed C compiler generates object files which use the following segments:

- _TEXT is the segment for executable code.
- _DATA is the segment for initialized data.
- _BSS is the segment for uninitialized data.

In addition, other segment names are used with a standard meaning. For example, the _INIT segment is used by the C start-up routine to access the initialization routines for library sub-systems. These segments are reserved. They must only be used for their allotted purpose(s).

Example : A Simple Function

The following example program returns a global variable in the AX register. Points to note are the declaration of the DATA and CODE segments of the program:

```
module Example

segment _DATA(DATA,28H)
public _i: dw 1

segment _TEXT(CODE,48H)
public _p:
    mov   ax,[_i]
    ret   0
end
```

The equivalent of this example in ‘C’ would be as follows:

```
int i = 1;
int p(void) {
 return i;
}
```

Tokens

The basic lexical tokens of TopSpeed Assembler are:

Generic Tokens	tokens relating to programmer selected items
Keywords	tokens relating to the control of the generated code
Instructions	tokens that are mnemonics for 8086 instructions

Neither keywords nor instructions can be used for user-defined names.

Generic Tokens

Generic tokens in TopSpeed Assembler are one of the following:

strings are sequences of characters enclosed in single quotation marks (‘).

numbers are signed decimal integers or hexadecimal numbers; hexadecimal numbers must start with a digit, all digits must be in upper-case and it must be suffixed with an H.

names are case-sensitive labels for segments, groups, subroutines and local jump destinations; they must begin with a letter (or an underscore, $ or @) and may contain letters, digits, underscores, $ or @. Some legal names are:

```
$loop
Main@Star           _again
```

registers which are the names of the 8086 registers (ax bx cx dx sp bp si di al bl cl dl ah bh ch dh es cs ss ds).

Keywords

The TopSpeed Assembler uses a number of keywords:

byte	db	dd	dw	dword
end	extrn	far	fixup	group
include	log2	module	near	org
power2	public	qword	section	select
seg	segment	st	tbyte	word

Instructions

The following 8086 instruction mnemonics are recognized by TopSpeed Assembler:

```
aaa fdiv  fptan jcnz  out
aad fdivp frndint je  pop
aam fdivr frstor  jg  popf
aas fdivrp  fsave jge push
adc ffree fscale  jl  pushf
add fiadd fsqrt jle rcl
and ficom fst jmp rcr
call  ficomp  fstcw jnc repe
cbw fidiv fstenv  jne repne
clc fidivr  fstp  jno ret
cld fild  fstsw jns rol
cli fimul fsub  jo  ror
cmc fincstp fsubp jp  sahf
cmp finit fsubr jpo sar
cmpsb fist  fsubrp  js  sbb
cmpsw fistp ftst  lahf  scasb
cwd fisub fwait lds scasw
daa fisubr  fxam  lea shl
das fld fxch  les shr
dec fldl  fxtract lock  stc
div fldcw fyl2x lodsb std
f2xml fldenv  fyl2xpl lodsw sti
fabs  fldl2e  hlt loop  stosb
fadd  fldl2t  idiv  loope stosw
faddp fldlg2  imul  loopne  sub
fbld  fldln2  in  mov test
fbstp fldpi inc movsb wait
fchs  fldz  int movsw xchg
fclex fmul  into  mul xlat
fcom  fmulp iret  neg xor
fcomp foption ja  nop
fcompp  fpatan  jbe not
fdecstp fprem jc  or

In addition to the above instructions, which are all unique, the following
alternative names are available for some instructions:
ja=jnbe     jne=jnz
jbe=jna     jp=jpe
jc=jb=jnae  jpo=jnp
je=jz     loope=loopz
jg=jnle     loopne=loopnz
jge=jnl     repe=repz=rep
jl=jnge     repne=repnz
jle=jng     shl=sal
jnc=jae=jnb xlat=xlatb
```

Operators

The following operators are defined in TopSpeed Assembler. The operators are listed in order of precedence (highest precedence first):

power2	power of two
log2	log base 2 (truncated to integer)
/ % *	division, modulus, multiplication
+ -	addition, subtraction
:	segment reference

~	bitwise not
&	bitwise and
|	bitwise or
seg	segment of address

Operators of equal precedence associate with the expression to their left.

Syntax

This section describes the TopSpeed Assembler assembly language. The conventions used are as follows:

::=	is used to mean “is defined to be”
|	is used to mean “or, alternatively”
.	is used to terminate a production of the syntax
<>	are used to enclose syntax elements
‘’	are used to enclose required typographic symbols

The syntax elements <*name*>, <*string*>, <*number*>, <*instruction*> and <*register*> are not defined here. The interpretation of these elements is described in the section *Tokens* earlier in this chapter. The syntax element <*empty*> is used to indicate that an alternative to a production may be omitted entirely.

- **<compilation> ::=**
  <globalsdefs> <modhead> <statements> end <entry>

```
· <globalsdef> ::=
    <globaldef>|
    <globalsdef> ‘;’ <globaldef> .
· <globaldef> ::=
    <name> ‘=’ <operand>|
    <empty> .
· <entry> ::=
  <empty> |
    ‘*’.
· <modhead> ::=
    module <name> .
· <statements> ::=
    <statement>|
    <statements> ‘;’ <statement> .
```

```
· <statement> ::=
    <instruction> <specifier> <operand-list> |(* Instruction *)
    db <byteexps> | (* Define Byte *)
    dw <wordexps>|(* Define Word *)
    dd <wordexps>|(* Define DWord *)
    fixup <exp>|(* generates fixup *)
    org <exp>|  (* reserve <exp> bytes *)
    <name> '=' <exp>| (* equate *)
    public <name> '=' <exp> | (* absolute public *)
    extrn <name>| (* external symbol/label *)
    section |   (* delimits minimum linker units *)
    select <exp>| (* selects new output segment *)
    segment <name> '(' <name> ',' <exp> ')' | (* segment *)
    group <name> '(' <segnames> ')' | (* group *)
    <label> ':' <statement> | (* label a statement *)
    <empty> .   (* empty statement *)

· <segnames> ::=
    <groupcomponent>|
    <segnames> ',' <groupcomponent> .
· <groupcomponent> ::=
  <exp> .
· <label> ::=
    public <name>| (* public (global) label *)
    <name> .(* normal (local) label *)
· <specifier> ::=
    <empty> |    (* default *)
    near    |
    far     |
    byte    |
    word    |
    dword |
    qword |
    tbyte .
· <byteexps> ::=
    <byteexp>|
    <byteexps> ',' <byteexp> .
· <byteexp> ::=
    <string>  |    (* outputs a string *)
    <exp> .     (* outputs a byte *)
· <wordexps> ::=
    <wordexp> |
    <wordexps> ',' <wordexp> .
· <wordexp> ::=
    <exp> .     (* outputs a word *)
· <operand-list> ::=
    <empty>   |  (* no operands *)
    <operand> |  (* one operand *)
    <operand> ',' <operand> .|(* two operands *)
· <operand> ::=
    <register>  | (* register operand *)
    <exp>   |  (* immediate operand *)
    <register> ':' <addr-list> | (* segment override *)
    <addr-list> | (* memory operand *)
    st '(' <exp> ')' |(* 8087 register *)
    seg <exp> .   (* segment of <exp> *)
· <addr-list> ::=
    <addr>    |
    <addr-list> <addr> .
```

```
· <addr> ::=
    '[' <exp> ']' |  (* displacement *)
    '[' <register> ']' . (* index register *)
```

- **<exp> ::=**
  <number> | (* decimal or hex *)
  <name> |
  '~' <exp> | (* bitwise NOT *)
  '-' <exp> | (* arithmetic negation *)
  <exp> '*' <exp>| (* multiplication *)
  <exp> '/' <exp>| (* integer division *)
  <exp> '%' <exp>| (* modulus *)
  <exp> '+' <exp>| (* addition *)
  <exp> '-' <exp>| (* subtraction *)
  <exp> '&' <exp>| (* bitwise AND *)
  <exp> '|' <exp>| (* bitwise OR *)
  '(' <exp> ')'|
  power2 <exp>| (* 2 ** <exp> *)
  log2 <exp> . (* log base 2 of <exp> *)

Assembly Language Considerations

This section groups together a number of points regarding the use of the TopSpeed Assembler. This section should be used in conjunction with Appendix D, which contains full details of the 8086 and 8087 instruction sets.

This chapter describes a number points of interest for the assembly level programmer, who is used to using other assemblers.

You may find it useful to use the TopSpeed Disassembler TSDA to generate examples of TopSpeed Assembler from high-level language object files.

The Symbol Table

The TopSpeed Assembler discards the current symbol table every time a new section is started, except for any global equates which occur *before* the module keyword in the source file. This means that symbols may be redefined. If you redefine a public symbol, the linker issues a warning message.

If you wish to reference external objects, the extrn keyword should be used, for example:

```
extrn ReturnOne
  ...
  call far ReturnOne
  ...
```

To make an object accessible from another module, the name must be preceded by the public keyword.

Operand Sizes

In most cases, the operands to an instruction have an implied size (byte, word, etc). This size can be calculated by the Assembler and requires no intervention on your part. However, when there is a chance of ambiguity, you must use a specifier to indicate the size of the operand:

```
inc word [bp] [-6]
mov byte es:[bx], 1
mov es:[bx], ax (* no specifier needed *)
              (* - ax implies word *)
push es:[bx]  (* push always uses a word *)
```

Jumps and Calls

Jumps and calls default to the smallest possible case. Thus, unless a label has been defined before use, the Assembler assumes that it is a label in the same segment as the statement referencing it.

Labels should result in a jump of between -128 bytes and +127 bytes in the output object code. If a jump exceeds this range, a specifier must be added. For example:

```
jmp near _loop
```

Strings

Single character string constants are treated as numbers by the TopSpeed Assembler.

The Assembler syntax does not provide a segment override for string instructions. If you wish to do this, it must be done using the db keyword. For example:

```
db 2EH; (*cs:*) stosb
```

References

Within TopSpeed Assembler, forward references are assumed for labels that have not yet been defined. The TopSpeed Assembler assumes that such labels are in the currently selected segment within the current section.

Within a single section, equated symbols may be redefined; labels, on the other hand, may not be redefined.

Non-8086 Instructions

The TopSpeed Assembler supports only 8086, 8088, 8087 and 80287 instructions. If you need to use the opcodes specific to the 80286, 80386 or 80387, they must be simulated with the db directive, as described in *Strings* above.

Floating-point Options

The foption mnemonic is used to control floating point emulation and carries a single argument, as follows:

`foption 0`	causes floating point emulation.
`foption 1`	suppresses the generation of emulation fixups.
`foption 2`	suppresses the generation of fixups and wait instructions.

Unless foption 2 is used, the Assembler generates a wait instruction between successive floating-point instructions in order to synchronize the 8086 with the 8087.

Variables and Data

Variables can be defined, and uninitialized data areas reserved, using the following directives:

`db`	initialize storage as bytes (expects byte or string operands).
`dw`	initialize storage as words (expects word operands).
`org`	reserve some uninitialized memory of a size equal to the value of the operand.

File Inclusion

The TopSpeed Assembler has an include directive which allows a header file to be included. This is used in the library to define commonly used identifiers:

```
include "corelib.inc"
```

Conditional Assembly

The TopSpeed Assembler has the ability to conditionally assemble sections of code based on the value of an identifier. There are three special directives that mark which blocks to include/exclude. These are:

(*%T <identifier> *)	means process this section only when <identifier>=1
(*%F <identifier> *)	means process this section only when <identifier>=0
(*%E *)	means end of conditional section

These directors are commonly used with the return code for a procedure that is memory model independent:

```
(*%T _fcall *)
ret 0
(*%E *)
(*%F _fcall *)
ret far 0
(*%E *)
```

Conditional sections can be nested.

For further examples of using conditional assembly, please refer to the file COREGRAP.A in the \TS\SRC directory. This file is also an excellent example of assembly language procedures providing memory model and register passed parameter support.

Predefined Identifiers

The assembler pre-defines a symbol for each #pragma define(<symbol>=><value>) which is active in the project file at the point where the #compile occurs. This allows conditional assembly according to which model has been selected. The value is 1 if the project value is on, otherwise the value is 0.

Memory Models and Predefined Identifiers

The following identifiers are defined by the Project System, and have the meaning shown.

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

Calling Conventions

Your assembly language functions need to follow the TopSpeed calling conventions if they are to successfully be called from a high-level language. In particular the bp register must be preserved. Which other registers need to be preserved will depend on the calling convention being used, which can be modified with pragmas. In the JPI calling convention, all registers except those used to pass parameters and return function results must be preserved.

Miscellaneous Points

Other assemblers allow instruction formats such as:

```
mov ax, es:[bx+6]
```

In TopSpeed Assembler, these must be expressed as:

```
mov ax, es:[bx] [6]
```

The instructions rep, repne and lock are regarded by the Assembler as separate instructions. They must be followed by a semi-colon, for example:

```
rep;  movsb (* NOTE the semi-colon *)
```

Smart Linking

When the TopSpeed linker links your program, it includes only the CODE and DATA segments which are actually referenced. In the following example _p and _q are placed in separate segments (of the same name) so that they will be ‘smart’ linked into the final program:

```
module Example2

segment _TEXT(CODE,48H)
public _p:
    mov   ax,1
    ret   0

segment _TEXT(CODE,48H)
public _q:
    mov   ax,2
    ret   0

end
```

TopSpeed Assembler Error Messages

This section summarizes the error messages, and their probable causes, which may be generated by the TopSpeed Assembler.

Bad char in decimal constant

Only digits are allowed in a decimal constant. The following are illegal:

```
x = 9A5
y = 99h           (* An upper case H is required for hex *)
```

Bad char in hex constant

The characters making up a hexadecimal constant must be in upper case. For example:

```
x = 0aaH
        (* is an error, aa must be AA *)
```

Bad character

The Assembler has encountered a character which it does not recognize, for example:

```
!x = 10
    (* '!' is not allowed in identifiers *)
```

Bad operand for group

The group keyword requires the names of previously defined segments as operands.

Bad operand for seg operator

The seg operator requires a label as an operand.

Bad operand for select

The select keyword requires a segment name as an operator.

Incorrect operand(s)

The operands to the instruction are of the wrong type. Check Appendix D: '*8086/8087 Instruction Sets*' for the correct format and type of the operands for the instruction in question.

Label not in expected segment

The Assembler has detected a forward reference to a label and has assumed that the label is to be found in the current segment. When the label was found, it was different segment. For example:

```
select  _TEXT
  ...
  mov ax, [x] (* forward reference *)
  ...
select  _DATA
x :
  dw 0      (* different segment *)
```

The solution is to move the definition of x to before its reference:

```
select  _DATA
x:
  dw 0
select  _TEXT
  ...
  mov ax, [x] (* Correct! *)
  ...
```

Mismatched comment

A comment (starting with '(*') has not been terminated before the end of the file.

Mismatched string constant

A string constant is missing the apostrophe (') terminator. For example:

```
db    'hello
```

Strings cannot be split across lines; they must be entirely contained on one program line.

Missing byte/word specifier

The instruction contains an ambiguity about the type of one of the operands. The Assembler cannot resolve the ambiguity and requires a byte, word or other specifier.

More than one label in address

An address expression can contain only one label. The following is an error:

```
x:    dw    0
y:    dw    0
    mov ax, [x] [y] (* Not allowed *)
```

No segment defined

At least one segment must be defined in a module before any instruction, label or data statements.

Not a segment register

A segment override in an expression must be one of the four 8086 segment registers (cs, ds, es, or ss). For example:

```
mov ax, dx:[bx] (* DX not a segment register *)
```

Not an index register

Only the registers bp, bx, di and si can be used as index registers. The following is therefore an error:

```
mov ax, [dx](* DX is not an index register *)
```

Operand may not be a label

The operands for arithmetic operators, org and equated symbols, must be pure numbers. If the Assembler encounters an undefined symbol it assumes it must be a label. This results in this error message in situations such as:

```
x = y + 5      (* y is not yet defined *)
y = 6
```

The solution is to exchange the order of the statements:

```
y = 6
x = y + 5      (* OK, y already has a value *)
```

Scope table full
Name table full
Too many identifiers

These messages are rarely seen. When they do appear, they indicate that your program file needs splitting up. Either create more than one module, or divide the module using the section keyword.

Syntax error

Part of the program does not conform to the syntax. There could be many reasons for this, for example:

```
db 0 0         (* missing comma *)
```

Refer to the syntax description of the Assembler for further information.

Token too big

This error is extremely unlikely. The maximum length of identifiers and strings is so ridiculously large that this message should not occur. If it does, you may have to shorten some identifier names.

Undeclared byte operand

A byte operand cannot be a label, so it must be declared before it is used. For example:

```
mov al, x      (* this causes an error *)
```

The correct way of using x in this context, is:

```
x = 99H        (* declare x with an equate *)
mov al, x
```

Unexpected end of file

An end statement is required at the end of the module.

Unknown/misspelled instruction mnemonic

The instruction is not known to the assembler. Check the spelling of the instruction and that you are not using a mnemonic (for example, dq) which is not supported by the TopSpeed Assembler.

Wrong number of operands

An instruction has too many, or too few, operands. Check Appendix D: '*8086/8087 Instruction Sets*' for the number and type of the operands for the instruction in question.

CHAPTER 7
POST-MORTEM DEBUGGER

The TopSpeed Post-Mortem Debugger is an adjunct to the Visual Interactive Debugger (VID). It allows you to examine a memory image of a program that is 'dead' from either a bug or a user-defined error. The Post-Mortem Debugger produces an image of the program state at the time a fatal error occurs. You can then use VID to examine this state in the context of the source code. Since this Post-Mortem Dump is a static image of the program's state at the time the error occurred, the VID facilities which can be used are restricted.

Overview

The basic concept of a Post-Mortem Dump goes back to the earliest days of computers, when system errors caused mainframes to produce vast listings of the "core image" of the computer's memory. You would then have to wade patiently through all this paper armed only with a manual of machine-language opcodes and the ability to do binary, octal or hexadecimal arithmetic, in an attempt to get the program going again.

To some people this remains the epitome of real programming, but others are glad that programmers eventually began to create tools to help each other.

Many tools became available to aid programmers in the debugging process. First came programs which interpreted the memory layout of the dump and attempted to convert some of the information to assembler-like mnemonics. Later came programs which enabled this process to be carried out interactively on a memory image stored on disk. Not only did this mean that the amount of paper was significantly reduced, but it also eliminated many simple errors on the part of the programmer analyzing the data.

More recently, fully interactive debuggers have become available, which allow similar functions to be performed on programs as they are running. VID is an example of such a development.

The use of such tools can have a significant effect on debugging and error correction. However, you must usually have a good idea which part of the program is causing the error. Another problem is that errors can occur in

programs after they have been released and users are not keen on running their applications under the somewhat technical eye of programs such as VID.

Using the Post-mortem Dump involves two steps:

- Adding the appropriate code to your source program and modifying your project file accordingly.
- Running VID on the resulting dump file (if an error occurs).

All fatal run-time errors, which you have modified your program to recognize, now cause your program to produce the dump file PM_DUMP.$$$ before your program terminates.

Including the Post-mortem Dump Facility in Your Programs

The inclusion of facilities for post-mortem debugging involves making small modifications to your source program and the associated .PR file. It also requires the use of one extra source file and the inclusion of a definition module into your program. This section describes the necessary changes in detail.

C source Code and Project Modifications

In order to use the post-mortem facility, the main file in the project must import the header file pmd.h:

```
#include <pmd.h>
```

The file pmd.h declares the relevant external functions and variables.

The post-mortem dump is enabled by inserting the following line into your main function:

```
includePMD=1;
```

Post-mortem dump may be disabled during execution by resetting the variable to 0.

The macro UserFatalError(code) is also defined to allow you to terminate the program and generate a post-mortem dump file. The example below causes the creation of a dump file and terminates the program with an error code of 14:

```
if(ErrorCondition)
  UserFatalError(14);
```

To link the required code, the file cpmd.c must be compiled and linked by inserting the following line into your project file:

```
#compile cpmd.c
```

Debug information must be present to allow VID to inspect the contents of the post-mortem dump file.

Modula-2 Code and Project Modifications

In order to use the post-mortem facility, one module in the project must import the definition file pmd.def:

```
IMPORT PMD;
```

The post-mortem dump is enabled during initialization of the PMD module, but may be disabled by resetting variable includePMD to 0:

```
PMD.includePMD := 0;
```

The procedure UserFatalError(code) is also defined to allow you to terminate the program and generate a post-mortem dump file. The example below causes the creation of a dump file and terminates the program with an error code of 14:

```
IF ErrorCondition THEN
  PMD.UserFatalError(14);
END;
```

Debug information must also be present to allow VID to inspect the contents of the post-mortem dump file.

Pascal Code and Project Modifications

In order to use the post-mortem facility, one module in the project must import the interface file PasPmd.it:.

```
IMPORT PasPmd;
```

The post-mortem dump is enabled during initialization of the PasPmd module, but may be disabled by resetting variable includePMD to 0:

```
PasPmd.includePMD := 0;
```

The procedure UserFatalError(code) is also defined to allow you to terminate the program and generate a post-mortem dump file. The example below causes the creation of a dump file and terminates the program with an error code of 14:

```
if ErrorCondition then
  PasPmd.UserFatalError(14);
```

Debug information must also be present to allow VID to inspect the contents of the post-mortem dump file.

Using VID with a Post-mortem Dump

In order to use VID to examine a post-mortem dump, simply start it with the following command:

```
VID /p
```

VID then searches the redirection list looking for the first file called PM_DUMP.$$$. If it cannot find this file, VID reports the error and terminates.

If PM_DUMP.$$$ is found somewhere on the redirection list, the file is loaded into memory, together with the appropriate source(s). You can then use all the normal VID facilities except, those concerned with changing the execution path of the program (i.e., setting breakpoints, etc) and actually running the program.

CHAPTER 8
WATCH

Watch is a *Terminate and Stay Resident* (TSR) program which allows you to monitor another program's use of DOS Function Calls. Watch does not require any extra instructions to be added to your program, and can be used with any DOS program that uses standard DOS function calls.

Watch may be used to debug programs written in any of the TopSpeed languages that generate code using DOS int21H services. When you have finished using Watch, you can easily remove it from memory.

DOS Function Calls

Watch monitors *DOS Function Calls*. Whenever a program runs under DOS, there are times when it uses the services offered by the operating system. (After all, one of the reasons an operating system exists is to provide common services to application programs.) Examples of some of these services are:

- A summary of how much memory this machine has.
- Open the file C:\WP\LETTER.DOC.
- Find the first file that matches the wildcard A*.COM.
- Read a line of 80 characters from the keyboard.
- Check the date and time.

You could do all of these directly by addressing your computer hardware. However, this would take time and knowledge, and would be terribly complex and worst of all, non-portable.

MS-DOS provides a set of subroutines that provide these services. You can imagine these subroutines as a built-in software library, heavily optimized for the hardware on which it is running. A call to one of these DOS services is known as a DOS function call.

Function codes

Each function is assigned a number, known as the *Function Code*. When your program wishes to perform a DOS function call, it first loads this function code into the 80*x*86's AH register, and then performs an int21H instruction, which calls the interrupt located at position 21H in the CPU's interrupt table. This interrupt table begins at the very bottom of RAM (address 0000:0000H).

In addition to the function number, certain DOS function calls also require parameters (for example, the Open File function requires the name of the file to be opened). These parameters are passed to the function call in a combination of the other registers of the 80*x*86 CPU. The particular combination of registers used and the format of their contents depend on the function being called. A complete list of all DOS function calls, together with their input parameters, can be found in Appendix C: '*DOS Function Calls*'.

As well as input parameters, the DOS functions also need to return a value (even if this is just to confirm success). This is also performed using the 80*x*86 CPU registers, primarily the AX register. A number of the functions indicate status information by setting and resetting the CPU flag registers. The primary flags which are used are Zero and Carry. The usage of output registers and flags is fully described in Appendix C: '*DOS Function Calls*'.

Calling DOS Functions

The following fragment of assembly code illustrates how DOS Function Calls are utilized in a program, regardless of the language in which that program is written.

```
....
mov ah,02H    (* Function code 02h *)
              (* Display output *)
mov dl,42     (*ASCII 42 = asterisk *)
int 21H       (*Call DOS *)
....
```

This fragment illustrates the information that Watch interprets when you use it to monitor a call. Watch gives you information about the contents of the 80*x*86 CPU registers, suitably interpreted for the call that the program is making. The advantage of this method is that you can see the context of the call, which, in turn, gives you the context in which an error is occurring.

Overview

Watch is a small program that sits in your computer's memory and monitors the DOS functions used by any other programs. You specify the DOS function calls that you wish to monitor, and, each time one of these functions is used, a window pops-up detailing exactly the environment in which the

call is being made. When the service has been completed, Watch displays a new window detailing the results (if any) which the function is returning to your program.

For example, when Watch is active and a program performs an int21H instruction, the following occurs:

1. The `AH` register is checked to see whether the called function is in the list of calls you have selected to be monitored.
2. If it is, Watch interprets the 80*x*86 register contents within the context of the call and displays a window (known as the *before window*) detailing the function called and the environment of the call. If this is not one of the calls you have selected, Watch takes no action and allows the call to proceed without hindrance.
3. Watch remains active in this way until you instruct it to restart by pressing the *SPACE* key. Control is then passed to the appropriate DOS service.
4. When the service is complete, Watch displays another window (known as the after window) and interprets the 80*x*86 register contents in the context of the returned values from the current DOS function call.
5. Pressing the *SPACE* key again returns control to your program exactly as if the call had proceeded without Watch's intervention.

While the before and after windows are active, Watch allows you to examine the environment of the call in more detail. You can:

- Examine the contents of your Program Segment Prefix (PSP), including the list of open file handles.
- Examine the contents of the 80x86 CPU's registers.
- Examine the current list of Memory Control Blocks (MCBs).
- Examine, using a formatted display, the contents of a File Control Block (FCB).
- Examine the contents of an I/O buffer (a record) that is being read or written.

Monitoring Errors

In addition to simply monitoring the progress of DOS function calls, Watch allows you to monitor their failure. "Fail" means any DOS function that returns a result indicating failure, or that is likely to result in an "Abort, Retry or Ignore?" prompt.

Watch can help in diagnosing the source of these errors, by displaying an explanation of the cause and allowing the environment of the error to be examined when the error actually occurs.

Requirements & Limitations

Watch is a powerful and useful program, but it requires that your computer, operating system and programs obey the rules. The requirements to run Watch are:

- Watch requires an IBM PC, XT, AT, PS/2 or closely compatible computer with a monochrome or color monitor (CGA, EGA or VGA).
- Watch does not run in graphics mode; it can only be used to debug programs (or portions of programs) where the computer is in text mode. This limitation only applies when the messages are being displayed on the screen; messages can also be sent to the printer or to a file.
- Watch can only monitor activities that properly use DOS function calls. Watch cannot be used to monitor a program if the program uses the computer's hardware directly (for example, writing directly to screen memory or explicitly using hard-disk controller registers).

Starting Watch

Watch can be started by typing the following command at the prompt:

```
watch
```

Watch is then loaded into memory, and you are presented with the initial menu and information screen. From this screen, you can select the function calls you wish to monitor and start the monitoring process. Full details of this main menu are given in the section '*Using Watch*', below.

Once Watch has been installed, it remains in your computer's memory until you explicitly remove it. Once Watch has started monitoring function calls it continues to do so until you either unload it from memory or switch off the monitoring. The monitoring can be switched back on again at a later time to allow you to continue with your debugging.

Running More Than One Copy of Watch

You can only run one copy of Watch at a time. When you attempt to load the program a second time, Watch detects that a copy of itself has already been loaded. If the loaded copy is not monitoring any function calls, the main menu appears in the normal way. If, on the other hand, you are currently monitoring a set of function calls, a special menu is displayed:

This menu has three options:

U - Unload Watch

When this option is chosen, the current monitoring session is terminated, and the monitoring file (if any) is closed. Watch is unloaded from memory if, and only if, it is the last program loaded. You cannot unload Watch if another program, which was loaded later, is still in memory; this would leave a "hole" in the MS-DOS memory map.

S - Show Monitored Functions

Selecting this option causes Watch to display a box containing a list of the functions currently being monitored. Press any key to return to the menu.

Q - Quit

Selecting this option returns you to the DOS command line without affecting the behavior of Watch.

Starting Watch with Parameters

As you become more familiar with Watch you will begin to learn the function numbers for individual functions. You can then set up the monitoring conditions, automatically, when you start Watch. This is done by specifying the codes as parameters on the command line. For example:

```
watch 06 09 0d
```

This command causes Watch to monitor the calls 06H (Direct Console I/O), 09H (Display String) and 0DH (Disk Reset).

Each code on the command line must be a two-digit hexadecimal number (i.e., you must type 03 and not 3). You can set up to 45 DOS function calls at any one time.

Using Watch

This section describes the Watch menus, options and related displays. The interface is simple, complete and powerful. It is designed to provide the maximum, manageable amount of information on your screen at a single time.

The fastest way to find out how Watch works is to use it to monitor a mature program. One course of action is to monitor some of the disk and/or character I/O functions while using the DOS command line. As well as demonstrating Watch, this gives some useful insights into the processes within COMMAND.COM when you type a command at the DOS prompt.

The Main Menu

Once you have started Watch, you are presented with a screen showing the Main Menu and a box listing the DOS function calls that are being monitored. This box is initially empty.

You can select an option from this menu in one of two ways, either:

- By moving the Item Highlighting Bar using the arrow keys, pressing RETURN to select the option, or:
- By typing the highlighted letter. In the case of the main menu the highlighted letters are: I, A, E, D, S and Q.

Most of these options pop-up another menu offering a further selection of options. The details of each Main Menu option are as follows:

I - Include an Entire Category

Within Watch, the DOS function calls have been grouped into 14 categories. Each of these categories covers a related group of functions, for example, Disk I/O or Character I/O. These groupings simplify the process of selecting the functions you wish to monitor by allowing functionally related calls to be accessed with a single keystroke.

The categories are displayed in a menu that pops-up when you select the I option. Each of the 14 categories are letter coded; A through N. To select a category, simply type the appropriate letter. A 15th category, *Functions that Fail*, is also available. This allows you to monitor only those functions that do not complete correctly. Most programs make many calls to DOS; hopefully your programs only contain one or two that do not work correctly. Watch allows you to ignore those DOS function calls where DOS is not reporting any error.

Note: The menu box is only 10 lines long, but there are 15 options. This means that you must scroll the menu up and down to display all the options. You can do this by using the cursor keys. You can also use *PgUp* to display the first 10 lines of the menu, or *PgDn* to display the last 10 lines

The list of DOS Function Call categories is:

A Character I/O

B Disk-level Services

C Directories

D Open/Create Files

E Close Files

F Read Files

G Write Files

H Find Files

I Delete/Rename/File-mode

J Date/Time

K Misc. DOS Services

L Program Start/End

M Memory Allocation

N Network Services

O Functions that Fail

The DOS function calls belonging to whichever category you select are added to the displayed list. You can choose as many categories as you like. Once you have finished selecting categories, press *Esc* to return to the main menu.

When you select a category, the screen is updated to show the list of functions you have chosen.

A - Add a Single Function

The use of categories is an advantage only when you are unsure as to which specific DOS function calls are causing the problem. Eventually, you should be able to narrow down your search to one or two calls.

Selecting this option pops-up a scrollable menu of all the DOS function calls, given in order of the function call number as listed in Appendix C. For each function call you wish to monitor, move the highlighted area over the function and press *RETURN*. When you have finished adding items, press *Esc* to return to the main menu.

E - Exclude an Entire Category

Selecting this option displays the same list of categories as the Include an entire category option.

For each category you wish to remove from your list, select the category by pressing the identifying letter. Again the list can be scrolled by using *PgUp*, *PgDn* and the cursor keys. Press the *Esc* key to return to the main menu.

Note: This function has no effect if there are no function calls in the list.

D - Delete a Single Function

Individual DOS Function Calls can be removed from the list by selecting this option. A scrollable list pops-up, as with the Add a single function option. To remove a call, select the function(s) you wish to remove and press *RETURN*. Pressing *Esc* returns you to the main menu.

As with the Exclude an entire category option, this option has no effect if there are no calls in your selection list.

S - Start Monitoring

Selecting this option starts the Watch monitor of the selected function calls. An Output Options menu appears asking you to specify where you want the messages to be displayed:

If you choose to send messages to your screen, the debugging information is displayed in a pop-up window. To continue running the program, you must then press a key. Displaying the results in this way, allows you to alter the current Watch configuration and examine the current environment of the monitored program.

If you choose to send messages to the printer or to a file, the before and after window information is written out as each DOS function call occurs. When using these methods, you have no opportunity to carry out interactive investigations as each call occurs.

Selecting the Send to file option prompts you for a filename. This file is then created in the current working directory (not in the directory where Watch was located). This file remains open until the current Watch session is completed, i.e., until Watch is unloaded from memory.

Q - Quit (Without Monitoring)

This option allows you to exit Watch without starting a monitor. The list of function calls to monitor is abandoned; you must reassign them before you can begin another monitoring session.

The Main Watch Windows

The Before Window

The *Before Window* is displayed before a DOS function is called. It describes the settings of the calling registers, showing the function being called and the time since Watch was loaded.

In the example below, the function which is being monitored is 0AH (Buffered Input). The parameters for function code 0AH are being passed in DS:DX. This contains the address of the buffer to be used for the input (DS has the segment; DX the offset). The first byte of the buffer is initialized to the size of the buffer (in this case 80H, 128 bytes). The second byte is reserved for the return (see below) of the actual number of characters received into the buffer. Below that is an image of the current buffer's contents (in this case, it is empty). Along the bottom of the window are listed the keys that are active. Pressing *SPACE* causes execution of the DOS function call to continue. The displays provided by the other keys are described in the following section.

The After Window

The *After Window* is displayed once the DOS function call is complete. The display is very similar to the before window.

The example below shows the after window of the previous example. The characters 'd' 'i' and 'r' have been typed on the keyboard, followed by the pressing of the *RETURN* key. The window looks like this:

The second byte of the buffer now has the value 03h, indicating that three characters are now in the buffer. The display shows those three characters. It also notes that the function completed successfully. The listed keys produce the same displays as they do in the before window, but with the information updated as necessary.

The Supplementary Windows

This section describes the supplementary windows available through the options displayed on the before and after windows.

F1 - Register Contents Display Window

Pressing *F1* from either the before or after window clears the window and displays the complete contents of the 80*x*86 CPU registers at the time the call was made.

Note: The contents of the registers are in hexadecimal and Watch makes no attempt to interpret them. Of course, some registers have meaning in terms of the call (in our example DS and DX are significant), while others simply contain "rubbish" or "nonsense" values. They may represent something in terms of your program, but this will depend on the source language, the compiler and the execution environment.

The diagram below shows a typical display:

The fields of the display are:

Event Count — shows the number of DOS function calls made in this Watch session.

CS:IP — shows the current *CODE Segment* and *Instruction Pointer* addresses.

Immediately below these two fields the other CPU registers are listed.

Flags — shows the uninterpreted state of the 80*x*86 flag register.

A full explanation of the 80*x*86's register architecture can be found in Chapter : '*TopSpeed Assembler*' of this manual.

Pressing *RETURN*, returns you to the before or after window.

F2 - Program Segment Prefix (PSP) Windows

Pressing *F2* from either the before or after window clears the window and displays the first of four sets of information about the program's *Program Segment Prefix* (PSP).

Every MS-DOS program has a PSP, which is an area of memory immediately before the program. The PSP contains two default File Control Blocks (FCBs), an image of the command line used to start the program, a table for referencing the first 20 open file handles and other miscellaneous information regarding the program. The following diagram is an example of the first display:

The display shows:

Process ID — MS-DOS assigns a sixteen-bit number (the Process ID) to every program that is run; this is the Process ID assigned to the program you are investigating.

Top of Memory — shows the last memory location used by the program under investigation.

Terminate — the address of the subroutine to which control will be passed when this program finishes.

Ctrl-Brk — the address of the subroutine to which control will be passed when the *Ctrl-Break* key combination is pressed.

Crit Err — the address of the subroutine to which control will be passed if a critical error occurs.

Parent PSP — the address of the Program Segment Prefix of the process which started the current program. In the case of programs started from the command line, this is the PSP of COMMAND.COM, but any program can be the parent process of another program.

Environ	the address of the current *environment variable block,* which is the area of memory where MS-DOS environment variables (such as PROMPT and PATH) are stored.

Pressing *RETURN* displays the first *File Control Block* (FCB). The two FCB displays are identical in format, but normally differ in their contents. The following example shows the first FCB:

In this example, the FCB is not in use and contains “rubbish” values. This is not unusual, since not all programs use the default FCBs in the PSP when accessing files. Pressing RETURN displays the second FCB for this program. Pressing RETURN again displays the *File Handle Table*:

This display shows the status of the first twenty file handles used by the program. Unless something unusual happens, the first five (0 - 4), are always open. These correspond to stdin, stdout, stderr, stdprn and stdaux. The remaining handles are available to the program.

Pressing *RETURN* displays the final screen about the PSP; an image of the command line parameters to the program. A typical display may look like this:

This display shows the contents of the command line buffer in hexadecimal on the left and in character form on the right. Notice that it is possible for the buffer to be filled with garbage. In this particular example, only the first 14 or so bytes are of any interest to the program.

Pressing *RETURN* returns you to the before or after window.

F3 - Memory Control Blocks Window

DOS keeps track of the memory occupied by each loaded program. This is the purpose of the *Memory Control Block* (MCB) table. Pressing *F3* from either the before or after window displays the current MCB.Using this table, you can see what programs are loaded, in what order, and how much memory they are using. A typical display might be:

For each memory block allocated, the table shows where the memory block starts, the address of the PSP of the program that owns that block and, if possible, the name or pathname of the program. Also shown is the size of the allocated block in bytes.

Pressing *RETURN* continues the list - if there are more entries to list - or returns you to the before or after window from which you began.

F4 - Currently Monitored Functions Window

The final display available from the before or after windows is a list of the DOS functions that Watch is currently monitoring. This display is obtained

by pressing *F4* from either window. The DOS function calls are listed by number; Appendix C contains a reference of the numbers and their associated functions.

A typical display might have the following form:

Using this screen, you can modify the functions being monitored. Press *F1* to add a new DOS function call to the list; press *F2* to delete one. You are then asked to supply the function code you wish to add or delete. Type this in hexadecimal, and press *RETURN*. The display is updated to show your modified selection list. Simply press *RETURN* to go back to the before or after window from where you started.

Suspending Watch

While Watch is active you can suspend it at any time by pressing *Alt-Backspace*. Pressing *Alt-Backspace* again re-starts Watch using the same DOS function call list for monitoring.

This facility is particularly useful if you want to re-compile your program. A compiler performs a large number of DOS function calls and Watch would, if not switched off, explain each one to you.

Watch for OS/2

The OS/2 version of Watch is very similar to the DOS version in its user interface. An initial menu allows the programmer to choose OS/2 Kernel API calls to be watched; they can be chosen individually or by category. Once the API calls are selected, an output medium is selected (screen-only, printer, or disk file). The target program is then executed as a child process of OS/2 Watch.

When a selected API call is made by the application program, both a before and after picture of the API call are shown. (If 'Calls That Fail' is chosen, only an after picture of the call is shown.)

At each before or after pop-up, the programmer can optionally look at the *Global Information Segment*, the *Local Information Segment*, and, if appropriate, a memory area associated with the API call (such as a Read Buffer). Also, the list of currently-watched API calls can be modified mid-stream. Finally, if desired, the process can be terminated.

Of course, OS/2 Watch (unlike the DOS version) is not a TSR. It runs the target program as a child process, and is invoked like this:

```
OS2Watch <progname> [prog parms]
```

As OS/2 Watch can not intercept “Int 21h”, and is unable to rename/replace DOSCALLS.DLL, it modifies the executable file to point to a special .DLL file (WATCHDOS.DLL), that contains code to show the parameters to each of the selected API calls. When OS/2 Watch finishes, it restores the DOSCALLS.DLL entry in the target executable. However, be aware - if the program aborts, it is sometimes possible that the WATCHDOS.DLL entry may get left inside the executable file. For this reason, you should make a practice of making a copy of the target executable file for OS/2 Watch purposes (or re-linking it after a session with OS/2 Watch).

CHAPTER 9
UTILITY PROGRAMS

This chapter describes the utility programs distributed with the TopSpeed TechKit©. These programs are the TopSpeed Disassembler (TSDA), the TopSpeed Profiler (TSPROF), the TopSpeed Module Utilities (TSMKEXP, TSIMPLIB and TSEXEMOD), the TopSpeed Help File Compiler (TSMKHELP) and the TopSpeed Executable File Compression Utility (TSCRUNCH).

File Redirection

All TopSpeed utilities use file redirection to locate their source and target files. See the " *TopSpeed User's Guide*" and the "*TopSpeed Developer's Guide*" for details of file redirection.

The TopSpeed Disassembler

The TopSpeed Disassembler (TSDA) produces an assembler listing from object (.OBJ) files. The listing is compatible with the TopSpeed Assembler and may be used to recover "lost" assembly language source files.

The Disassembler is invoked from the command line with the following form:

```
TSDA <object-file-name>
```

If you do not specify an extension to the file name, .OBJ is assumed. TSDA issues an error message if it cannot find the file. Directories specified in the default redirection file will be searched.

The Disassembler has a *source line include facility*. If an object file contains line number information, the disassembler will insert the lines from the source file into the output. This makes it much easier to determine which code has been produced from which source line.

To produce an object file containing source line information, the pragma debug(line_num=>on) or the command line option /b must be specified when the object file is compiled. In addition, the object file must be compiled with either minimum or full debug information. This can be done

in two ways, either by using the pragma debug(vid=>min) or debug(vid=>full), or by using the command-line option /v1 or /v2.

Note: If you are disassembling object files for the purpose of comparing the quality of generated code, you must specify (vid=>off). If full debug is specified, the code which is generated is considerably less efficient. This is because values are kept in memory rather than registers, so that VID can read and/or modify them. If minimum debug is specified, the code which is generated is close to the quality of the code generated without debug information. However, certain optimizations concerning the re-ordering of jumps, are not performed. This is because the correlation between machine code and source code lines would become obscured to the point where VID could not operate.

The assembly listing is directed to standard output, which defaults to the terminal screen. If you wish to redirect the output to a file for later editing, you should use the MS-DOS redirection facility. For example:

```
TSDA myprog.obj >myprog.a
```

This command disassembles the object file myprog.obj and produces a TopSpeed Assembler listing in the file myprog.a.

The Disassembler can be put to good use in tracking down problems in object files produced by other compilers. You can use the Disassembler to correct these problems and use your favorite utilities with TopSpeed. You may even wish to use it to examine and compare the relative efficiency of the code generated by various compilers.

TopSpeed Program Profiler

The TopSpeed Program Profiler (TSPROF) allows you to create an execution profile of your program. This execution profile shows the frequency with which different parts of your program are actively being executed. With this information, you can direct your attention to improving of your application in the areas where it is most effective.

TSPROF loads and runs your program and produces a file containing an analysis of what happened while it was running. In order to achieve this you need to generate a .MAP file when you make your program. You should also set the line number option (#pragma debug(line_num=>on) when compiling all modules that are required to be profiled (see the *"TopSpeed Developer's Guide*" for details).

TSPROF operates by sampling your program about 1000 times a second, and by determining the point the program has reached every time it takes a sample.

The information produced by TSPROF can be generated in two formats. The default format produces a list showing the count of the number samples for each function called by the program. The second format, which requires the /L option, produces a list showing the line number the program had reached at the time of the sampling.

If TSPROF cannot identify the line number or the function, it prints the word Unknown. If the memory area sampled is outside your program, TSPROF prints, where possible, the owning program's name.

The syntax for using TSPROF is:

```
TSPROF { /<option> } <progname> [ <progparams> ]
```

The following options are defined:

/O	followed by the name of a file. This option changes the name of the file where the profile is produced. The default is <progname>.PRF.
/L	causes TSPROF to generate a profile based on the source file's line numbers. If this option is not used then the profile is based on the functions called. In order to use this option you must have compiled the program's modules to include the line number information.

An Example

This example shows a profile of the sieve program. Having generated a .MAP file at link-time and compiled the program with line numbers, the following command line can be used to profile its execution:

```
C:>TSPROF /L sieve
```

This results in the following display on the screen:

```
TopSpeed Profiler                Version 3.00
Copyright (C) 1988 Jensen & Partners International
Reading mapfile SIEVE.MAP
Starting profile, Output file SIEVE.PRF
50 iterations

1899 primes
C>_
```

The first four lines are printed by TSPROF, the others by sieve.

The file SIEVE.PRF contains the following information:

```
Execution Profile for SIEVE.EXE
===============================

Program base at segment 1B67

Seg :  Range     Symbol name        Samples   %
-------------------------

027C:14D2-6707   Unknown (DOS)              21
09C7:00AD-03EB   Unknown ()                  8
0A2A:0149-0149   Unknown ()                  2
12E0:08DC-08DC   Unknown                     1
1B67:0088-008C   SIEVE.C     Line 18    711  11
1B67:008D-0091   SIEVE.C     Line 17    384   6
1B67:0098-009E   SIEVE.C     Line 22   1043  17
1B67:009F-00A5   SIEVE.C     Line 23     88   1
1B67:00A6-00AB   SIEVE.C     Line 24     98   1
1B67:00AC-00B0   SIEVE.C     Line 26   1323  21
1B67:00B1-00B6   SIEVE.C     Line 25   1660  27
1B67:00B7-00BC   SIEVE.C     Line 21    698  11
1B67:00D1-15C7   SIEVE.C     Line 40          5
F000:D90C-F09A   Unknown (BIOS)             6

Sorted
------

1B67:00B1-00B6   SIEVE.C     Line 25   1660  27
1B67:00AC-00B0   SIEVE.C     Line 26   1323  21
1B67:0098-009E   SIEVE.C     Line 22   1043  17
1B67:0088-008C   SIEVE.C     Line 18    711  11
1B67:00B7-00BC   SIEVE.C     Line 21    698  11
1B67:008D-0091   SIEVE.C     Line 17    384   6
1B67:00A6-00AB   SIEVE.C     Line 24     98   1
1B67:009F-00A5   SIEVE.C     Line 23     88   1
027C:14D2-6707   Unknown (DOS)              21
09C7:00AD-03EB   Unknown ()                  8
F000:D90C-F09A   Unknown (BIOS)             6
1B67:00D1-15C7   SIEVE.C     Line 40          5
0A2A:0149-0149   Unknown ()                  2
12E0:08DC-08DC   Unknown                     1
                                        ------

                Total samples        6048
```

The information is shown in two forms:

- The first list shows the sampling information ordered by memory location. The total number of samples for each area are shown (together with the location in the program) and the percentage of the total that this sample represents. Percentages less that 1% are not shown.
- The second list shows the list sorted by descending order of frequency. This makes it easy to see the most “popular” lines of your program.

  TSPROF runs under MS-DOS only. It does not run under OS/2 nor does it run in the DOS Compatibility Box under OS/2. TSPROF needs to be able to successfully commandeer the MS-DOS timer interrupts to ensure that sampling functions correctly. OS/2 disapproves of this so much that TSPROF causes an OS/2 system to “hang” if you attempt to run it, even in the DOS

compatibility box.

TopSpeed Module Definition File Generator

The program TSMKEXP can be used to generate an export list from either an object or library file:

```
TSMKEXP infile[.obj|.lib] expfile
```

Directories listed in the default redirection file will be searched.

TSMKEXP will output a file containing an export list of all the public symbols from the binary object or library file in the following form:

```
EXPORTS
  Func1    @?
  Func2    @?
  ....
```

Note: When making DOS DLLs a default segmentation setup must be included in the file. See Chapter 2: '*Segment-based Overlays*' for more information.

TopSpeed Import Library Generator

The program TSIMPLIB can be used to generate an import library from a module definition file:

```
TSMKEXP libfile.lib expfile.exp
```

Directories listed in the default redirection file will be searched.

See Appendix B: '*Module Definition File Syntax*' for more information.

TopSpeed Module Header Utility

The program TSEXEMOD can be used to modify the header and segment information in a new format executable file (.EXE or .DLL), using the information in a module definition file:

```
TSEXEMOD binfile.* expfile.exp mapfile.map
```

Directories listed in the default redirection file will be searched.

See Appendix B: '*Module Definition File Syntax*' for more information.

TopSpeed Executable File Compression Utility

The TSCRUNCH program is is a utility to compress .EXE files to the minimum possible size under DOS. The executable file is compressed using LWZ compression techniques and then has a small loader/expander added to the startup code. This form of compression can reduce the exe file by up to 50% of its original size.

To use TSCRUNCH just type the following at the DOS prompt:

```
TSCRUNCH exename
```

For example:

```
tscrunch myprog
```

This will create a new executable file (providing the compression is worthwhile) and create a backup of the uncompressed file with the extension .EXB.

It should be noted that compression of an exe file will cause the loading time to greatly increase, so it is best used on .EXE files that are either infrequently used or where the loading time is not significant.

Note: TSCRUNCH cannot be used for overlay or dynalink model programs, nor for Windows or OS/2 programs. TSCRUNCH cannot be used for files containing VID or CodeView debug information.

TopSpeed Help File Compiler

You can use TSMKHELP to make your own help files for use within the TopSpeed environment. Help files are compressed files with the extension .HLP and are generated from specially formatted text files (with the extension .HTX). To run TSMKHELP just type the following at the DOS prompt:

```
tsmkhelp <helpname>
```

and <helpname>.HTX will be read and <helpname>.HLP generated.

The file TSMAIN.HTX is supplied as an example .HTX file which allows you to re-make the ‘root’ TopSpeed help file. The format of the .HTX file is described in the supplied file TSMKHELP.DOC.

Chapter 10
TSR Support for Modula-2 Programmers

The TopSpeed TechKit© contains all the definition and implementation files you need to create a *Terminate and Stay Resident* (TSR) module in Modula-2. This module enables you to create memory resident programs. When first run, these programs execute, terminate and then remain in memory - as an extension to the part of DOS that remains resident - in an inactive state until invoked through a special “HotKey” sequence.

Most TSR programs, and all those written using this module, go through the following steps when run:

- The interrupt services table is redefined so that the entry for the keyboard interrupt points to a short routine. This routine compares your keystrokes to the hot-key sequence and activates the program when a match occurs. Where no match occurs the keystrokes are passed directly on to DOS.
- The size of the routine to remain resident is derived, along with any stack and heap space needed. This figure (rounded up to the nearest paragraph) is used when calling DOS to terminate and keep the program.
- DOS function 31H is called, requesting termination of the current program, freeing all memory other than the required resident portion.

The code for a TSR program remains resident in a portion of working memory, until the HotKey is pressed. At this point the program is activated and carries out its task. The program then suspends itself (until the next HotKey press) and execution continues in the program that had been interrupted.

Once a TSR has been correctly installed, the memory organization of your PC looks similar to this:

```
Interrupt Services Table
Resident hidden DOS programs (IBMBIO.COM & IBMDOS.COM)
COMMAND.COM
Resident part of your TSR, size allocated with termination call
Section of your program that installed the resident part (This may be
overwritten by any other programs)
Rest of memory
```

To help you to understand how a TSR module works, the source code is provided with your software in the file TSR.MOD. However, be careful if you intend to modify it.

Note: When writing TSR programs you must ensure that the model is defined as mthread. The reason for this is that the TopSpeed TSR module uses the Modula-2 *Transfer* facility to activate TSRs; this facility is available only with this model.

Activating a TSR Program

The HotKey which activates the TSR program is usually a combination of one or more shifted keys (*Ctrl*, *Alt*, *RShift*, and *LShift*). The normal keys may also be used in combination with the shift keys.

When a shift key is pressed, a bit is set to ON in a status mask at a particular address. Where a program requires a further key to be pressed, this returns a scan value associated with that key. This combination of status mask and scan values constitutes the HotKey to activate the TSR program.

In the TopSpeed TSR module, the status mask bits are represented in the following set:

```
TYPE
  KBFlagSet =
      SET OF ( RShift, LShift, Ctrl, Alt,
               (* Available shifts *)
                       Scroll, Num, Cap, Ins );          (* Unavailable *)
```

Only the first four keys specified in a KBFlagSet are available for use in HotKeys built with the TSR module.

Note: When writing a TSR program, you must specify the mask and scan values to make up your HotKey. These values are passed as parameters to the procedure Install, which is contained in the TSR module.

Installing a TSR Program

Before you can activate a TSR program, you must install it in memory. The Install procedure does this for you.

The declaration of Install is as follows:

```
PROCEDURE Install (P       : PROC;
                   KBF     : KBFlagSet;
                   Scan    : SHORTCARD;
                   heapsize: CARDINAL
                  );
```

P must be a global procedure. It is called whenever the TSR program is activated.

KBF — is a set of shift keys which should be pressed for the HotKey.

Scan — is the additional scan code. (Table 4-1 shows the scan codes for the IBM-PC.) If zero is specified for Scan, then only the shift keys need to be pressed.

heapsize — is the minimum amount of heap your program requires (in paragraphs).

For example, Install could be called as follows:

```
Install( TheAction, KBFlagSet{ LShift, Ctrl},30, 256);
```

This call specifies:

- The TSR program is activated by pressing Ctrl-LShift-A.
- The program needs 4K of heap space (16 * 256).
- When activated, the program calls the procedure TheAction to perform the allotted task.

Install does not complete until the TSR is de-installed (that is, the program is unloaded from memory).

Scan Codes for the IBM PC

Key	Scan	Key	Scan
Esc	1	1-9	2-10
0	11	-	12
=	13	BackSp	14
Tab	15	Q	16
W	17	E	18
R	19	T	20
Y	21	U	22
I	23	O	24
P	25	[	26
]	27	CR	28
A	30	S	31
D	32	F	33
G	34	H	35
J	36	K	37
L	38	;	39
‘	41	\	43
Z	44	X	45
C	46	V	47
B	48	N	49
M	50	,	51
.	52	/	53
*	55	SP	57
F1-F10	59-68	Home	71
End	79	PgUp	73
PgDn	81	Del	83
Minus	74	Plus	78

An Example TSR Program

The following listing contains a very simple TSR program. This program writes “Hello” whenever *Ctrl-LShift-A* is pressed:

```
MODULE MyTSR ;

IMPORT IO,TSR ;
PROCEDURE Activate() ;
BEGIN
  IO.WrStr('Hello');IO.WrLn;
END Activate ;

BEGIN
  TSR.Install(Activate,
        TSR.KBFlagSet{TSR.LShift,TSR.Ctrl},
        30,
        128);
END MyTSR.
```

Cautions about TSR Programs

Great care must be taken when writing TSR programs. You must remember that DOS and the BIOS are not re-entrant; they only have one set of variables. This means that if a program has just called MS-DOS, and your TSR pops-up and tries to use the same MS-DOS variables, the result may become rather confused, and the system will probably crash.

The TopSpeed TSR module avoids this by limiting your TSR to (safely) calling interrupt 21H functions 01H to 0CH only.

Other considerations that you must take into account when creating TSR programs are:

- You cannot use DOS functions below 0EH. This includes IO.RdKey and IO.KeyPressed.
- You cannot use the scheduler provided in the TopSpeed Process module.
- TSRs must be created with the multi-thread model.
- You must ensure that sufficient heap and stack space has been allocated (see below for how to do this).
- You should switch-off run-time checking. If a run-time error does occur, the machine will require a hard reset.
- If you plan to use more than one TSR at any one time, they may clash and require loading in a particular order. You may need to experiment with this.
- When a TSR program starts to work, it does not check to see what is displayed on the screen before starting to write. This

may corrupt the screen. To avoid this, a window should be opened to house the TSR program's input and output.

- A good programming point - if you intend to write a TSR program, ensure that it is well behaved towards system calls, interrupts and resources. This minimizes clashing with other programs.

TSRCALC.MOD provides a more complete example of a TSR program. This program makes use of a window for calculations.

Note: The window is defined in the main TSRCALC program, not in the procedure RunCalc (this executes when the TSR program is activated). RunCalc simply uses the window for its screen I/O.

If insufficient heapspace is allocated (i.e., the heapspace parameter is too small) you will prevent the program from being loaded. The following error message is produced:

not enough storage

It may require some trial and error until you find a heapspace that is small enough not to steal unnecessary memory, but large enough to handle your TSR program safely.

The size of the heap before the call to TSR.Install is not relevant, since the remaining heapspace is freed by the TSR.Install procedure.

The stack can be set using a compiler pragma, in this case:

```
(*# data(stack_size => n)*),
```

This should be located in the main module and allocates n bytes for the program's stack.

The TSR program should not be run from inside the TopSpeed Environment. When control is returned to the environment, the storage where the TSR program is located is reclaimed, which crashes your machine.

Deactivating a TSR Program

When the TSR program has finished its execution cycle, it must return control back to the interrupted program. This deactivation process stops the TSR program but does not remove it from memory. Rather, the program is suspended until it is activated again.

The example above deactivates itself immediately after writing "Hello" on the screen. You need not do anything for this to happen. Other programs (for

example TSRCALC) continue to run until you indicate that you have finished. Generally, pressing *Esc* accomplishes this.

Terminating TSR Programs

It is also possible to terminate a TSR program, which removes it from memory. Unlike deactivation, which just suspends the program, termination actually frees the memory in which the program had been residing.

To terminate a TSR program, you must call Deinstall or HALT. Such a call restores all interrupt traps, and frees the memory that had been allocated to that TSR program. To accomplish this, simply include the following statement at the appropriate point in your program:

```
Deinstall;
```

It is very important to terminate TSR programs in the correct order in a stack-like fashion. That is, the last TSR program installed must be the first one to be terminated. Ordinarily, you must be at the command prompt to Deinstall your TSR programs. Thus, if a program is still active, it must be terminated first.

Note: The simple TSR program used earlier does not call Deinstall. This means that the program remains resident until you reboot. Other programs allow you to terminate them. For example, pressing *Alt-X* while TSRCALC is active terminates the program. Pressing *Alt-X* produces a call to Deinstall.

If your TSR program cannot guarantee (to itself) that it is the last program in the memory, then it may be wise to prevent yourself from being able to terminate the program.

The TSR Module

The TSR module is interesting for several reasons. It represents a good example of the advantages of keeping procedures and data hidden, within an implementation module.

The definition part of the module is quite sparse. There are only two procedures which are defined, these are Install and Deinstall. The only data structure is the KBFlagSet. This module is very easy to read and understand.

The implementation module, on the other hand, contains 15 procedures and several other data structures, as well as numerous constants. Setting up a TSR program can be quite messy, but this is hidden in the implementation module.

The TSR module is also useful as an introduction to handling interrupts and manipulating storage in a DOS environment.

The TSRCALC Program

The sample TSR program, TSRCALC, is a simple, four function calculator for integer computations. Once you have compiled the calculator, you can install it by typing:

```
TSRCALC
```

Once installed, you can activate the calculator by pressing *Alt-Z*.

When you activate the calculator, the program opens a small window in the center of the screen. Any numbers that you enter are echoed in the window; operators are not. You can do your computations in base 2, 10, or 16. Check the source code for the RunCalc procedure for more information on what TSRCALC allows you to do.

When you have finished using TSRCALC for a session, press *Esc* to suspend it. To remove TSRCALC completely, press *Alt-X* while TSRCALC is activated.

CHAPTER 11
RS-232 SUPPORT FOR MODULA-2 PROGRAMMERS

RS-232 Support

The TopSpeed Modula-2 rs module provides the procedures and data structures that you need to make use of your computer's RS-232 serial communications port. The module definition contains several procedures for configuring, using and monitoring this port. You can use either communications port 1 or 2.

Serial communication generally takes place asynchronously, that is, one event dictates the timing of another. This is in contrast with synchronous communications (usually used for communications to mainframes and some LANs) in which each event occurs at a certain time, normally dictated by a clock pulse.

Rather than tying up the computer waiting for an asynchronous event to occur, communications programs are usually 'interrupt-driven' - that is, they act upon an interrupt from the port. The serial port hardware generates an interrupt whenever data has arrived at the input buffer or has been correctly sent.

An alternative method of operation is for the communications program to continuously check the input buffer, to see if anything has arrived. This method is called *polling* and is not used in this module, since it is really only suitable for slow data rates.

This means that interrupts play an important role in the process. Interrupts are particularly important when multiple processes are running at the same time. In this case, interrupts are used not only to coordinate the scheduling of processes, but also to control the use of buffers and storage.

For example, a communications program might disable certain types of interrupts until the entire contents of a buffer were transmitted. This would prevent another process from inadvertently destroying the integrity of a packet of information, by writing between elements of the packet.

The rs module works by installing new interrupt handlers that signal the status of the serial port, i.e. whether data can be sent, has arrived or if it is empty or full.

The rs module is designed to handle communication ports 1 and 2 on your computer. However, the source can easily be modified to provide support for ports 3 to 8, if necessary.

To correctly drive COM1 or COM2 you must know the interrupts they use and the port addresses they occupy. These are:

Port	Interrupt	Address
COM1	4	3F8H
COM2	3	2F8H

It is useful to know that the computer's BIOS provides a call to access COM1 and COM2 in the form of int 14H. However, this is far too slow for any reasonable speed of communications (1200 baud and over). When you need speed, use the rs module.

The rs.def File

The definition part of the TopSpeed rs module defines two data types:

`pt`	the parity being used
`wl`	the number of data bits being used in the transmission

These are used by a number of the procedures in the module.

Initializing the Port

The rs module uses two separate procedures to set up the hardware to allow proper use of the port.

The Install Procedure

The Install procedure installs the appropriate interrupt handlers for the specified port. This procedure must be called before any of the other rs procedures can be called.

The declaration of this procedure is as follows:

```
PROCEDURE Install(Port:CARDINAL);
```

where Port is the number of the port to be initialized. For example:

```
Install(2)
```

This program statement installs the necessary interrupt handlers, to allow a user to work with the second communications port, COM2:

The Install2 Procedure

The Install2 procedure is a secondary installation routine in which the interrupt to be handled must be specified (using the Intr argument). This is mainly to allow programs to access communication ports other than COM1 and COM2 (or any other non-standard serial port).

The declaration of this procedure is as follows:

```
PROCEDURE Install2(Port,Intr:CARDINAL);
```

For example, on a number of machines COM3 and COM4 can be used as follows:

For COM3:

```
Port3[40H:4]:Cardinal;
Install2(Port3,4);
```

and for COM4:

```
Port4[40H:6]:Cardinal;
Install2(Port4,3);
```

The Init Procedure

This procedure initializes the port specified by Install and sets the port parameters according to the specified settings passed as arguments.

The declaration of this procedure is as follows:

```
PROCEDURE Init(Baud  : CARDINAL;
          WordLength : wl;
          Parity     : pt;
          OneStopBit : BOOLEAN;
          HandShake  : BOOLEAN);
```

The parameters have the following ranges:

`Baud`	the baud rate, this can be up to 115,200 on fast machines.
`WordLength`	specifies the number of databits in each packet. This must be between 5 and 8 bits.
`Parity`	specifies the parity checking (if any) to be used. This may be: `None`, `Even`, `Odd`, `Mark` or `Space`.
`OneStopBit`	a `BOOLEAN` switch, either one stop bit or none.
`HandShake`	a `BOOLEAN` switch setting CTS handshaking on or off.

The BOOLEAN values define whether each packet ends with a single stop bit, and whether the two computers are to use *handshaking* to coordinate

their communications. (The handshaking used is hardware-based, using the RS-232 CTS line.)

For example, the following statement initializes communication at 1200 baud, using seven data bits and one stop bit, with even parity, and handshaking:

```
Init(1200, 7, Even, TRUE, TRUE);
```

Keeping Track of Buffers

Buffers are set-up to handle the received and transmitted data. These are initially set to 256 bytes in length, but can be increased by modifying value of BufferSize in RS.MOD. There are three procedures supplied which enable you to determine information about the state of the buffers used in receiving or sending information.

The RxCount Procedure

This procedure returns the number of bytes in the receiving buffer. This procedure is declared as follows:

```
PROCEDURE RxCount(): CARDINAL;
```

For example:

```
VAR NrIn : CARDINAL;
...
NrIn := RxCount();
```

The TxCount Procedure

This procedure returns the number of bytes still to be sent, i.e. those remaining in the transmitting buffer. This procedure is declared as follows:

```
PROCEDURE TxCount():CARDINAL;
```

For example:

```
VAR NrLeft : CARDINAL;
...
NrLeft := TxCount();
```

The TxFree Procedure

This procedure returns the number of bytes free in the transmitting buffer. This buffer is particularly important in a program which uses the RS-232 interface to transmit packages of information, so that you can ensure that a package is not split up during the transmission process. This procedure is declared as follows:

```
PROCEDURE TxFree():CARDINAL;
```

For example:

```
VAR NrFree : CARDINAL;
...
NrFree := TxFree();
```

Sending and Receiving Data

The two procedures that actually exchange information with other programs are Send and Receive. Both procedures have a length parameter, which specifies the number of bytes to send or receive.

Information about the number of elements in the buffer is important when using these procedures. For example, if the length parameter specifies that more bytes are to be sent than are free in the transmitting buffer, then the system may do a *Busy Wait*, thereby tying up the system, unless preventative actions are taken. The same thing would happen if a call to Receive expects to receive more bytes than are in the receiving buffer.

The Send Procedure

This procedure sends a specified number of bytes to the port being used. The information being sent is held in a specified buffer. This procedure is declared as follows:

```
PROCEDURE Send ( Buf : ARRAY OF BYTE; Len : CARDINAL );
```

where Buf is the name of the buffer and Len is the length in bytes.

For example:

```
VAR XBuff : ARRAY [ 300] OF BYTE;
...
Send ( XBuff, 200);
```

sends 200 bytes, taken from the array XBuff. These bytes are sent to whichever port (1 or 2) is being used.

The Receive Procedure

This procedure receives a specified number of bytes through the port being used. The information being received is stored in a specified buffer. This procedure is declared as follows:

```
PROCEDURE Receive ( VAR Buf : ARRAY OF BYTE;
                    Len : CARDINAL );
```

For example:

```
VAR XBuff : ARRAY [ 300] OF BYTE;
...
Receive ( XBuff, 200);
```

receives 200 bytes, and stores them in the array XBuff.

Sending and Checking for Breaks

The following procedures allow you to send a break signal from one computer to the other, and to test whether a break signal has been received by your computer.

The Break Procedure

This procedure sends a break signal lasting of a defined period in milliseconds.

This procedure is declared as follows:

PROCEDURE Break (Time : CARDINAL);

For example:

Break (500);

sends a break signal for 0.5 seconds.

This should only be done when the send buffer is empty, which can be established with a call like:

```
WHILE TXCount()#0 DO END;
Break (500);
```

The BreakTest Procedure

This procedure returns TRUE if a break signal has been received, and FALSE otherwise. This procedure is declared as follows:

PROCEDURE BreakTest(): BOOLEAN;

An Example Program

The following, short program demonstrates the instructions outlined above. This example initializes COM1 to communicate at 1200 bits per second, and will display everything received until any key is pressed.

The program then sends a break signal.

```
MODULE CommsExample
IMPORT rs,IO,Str,Lib;
VAR inp : CHAR;
BEGIN
  rs.Install(1);
  rs.Init(1200,8,rs.Even,TRUE,FALSE);
  (* sets 1200bps, an 8-bit word, even
     parity, 1 stop-bit, no handshaking *)
  LOOP
    IF IO.KeyPressed() THEN
       (* see if a key has been hit *)
      rs.Break(1000);
         (* send a one-sec break signal *)
      EXIT
    END;
    IF rs.RxCount()#0 THEN
      rs.Receive(inp,1);
      IO.WrChar(inp);
       (* If the i/p queue is empty, then
          receive one character and write it to
          the screen *)
    END;
  END;
END Example.
```

The RSDEMO Program

The RSDEMO program included in your TechKit© is a simple driver for the RS-232 port. The program uses the rs module.

RSDEMO can dial a number for you, and can send the appropriate string to start communications with another computer. Once communications have been established, the program transmits anything you type at the keyboard, until you press *F2*. The program hangs up when you press this key.

In its supplied form, the program does relatively little checking and safeguarding. You may wish to make the program more sophisticated and robust.

CHAPTER 12
ADVANCED LIBRARY USAGE

Library Initialization and Termination

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 - 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 or static destructors are 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 unless an exception, such as a segment over-run, occurs.

- 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.

Note: For information concerning creating initialization code in assembler see Chapter : '*Multi-language Programming*'.

The Library and Embedded Systems

Embedded system users will not be able to use the standard TopSpeed startup code, or all of the modules and functions that comprise the standard libraries. The following section gives a broad indication of the approach that an embedded systems programmer should take.

Startup

The startup file INITESYS.A should be used in place of INITEXE.A. This file should be used as a basis for program startup; individual library files may be linked as required.

Linking Library Files

The module CoreRtl should be used to provide support for compiler generated calls and Modula-2/Pascal initialization. The choice of other modules used depends on the exact nature of the run-time environment.

C Library Considerations

Using the C run-time library in an uninitialized state the following groups of functions and variables are not available or are subject to restrictions:

- Command line arguments. The global variables _argc and _argv are not available.

- Post-mortem dump: not available.
- Signal handling. The functions abort, signal and raise are not available.
- The program timer. The functions clock (time.h) and delay (stdlib.h) are not available.
- File IO.
- All functions using FILE structures are not available, but the string formatting functions such as sscanf and sprintf may be used subject to the restrictions below.
- The standard files stdin, stdout, stderr, stdprn and stdaux are not available except to those low level I/O functions with preceding underbars (for example, _read, _write etc) which do not use the file descriptor table.
- All functions using file handles may be used on files other than the predefined handles. For example open, close, read, write etc.
- Process sub-system. Multi-thread programs may not be used (process.h). Child processes may not be spawned using exec??, spawn?? and system.
- Floating point. Floating point can be present, but the exception handling mechanism will not be initialized causing library math functions to behave incorrectly. It is therefore not advisable to use floating point functions.
- Memory allocation. Neither the far or near heap functions are available.
- Text windows. Window functions (window.h) may not be used.
- Graphics. Graphics functions (graph.h) may not be used.
- Environment strings. The local C environment is not available so functions putenv, getenv and searchenv may not be used.
- String handling. strdup may not be used since it calls malloc.
- Console I/O. All console I/O functions can be used, but not in conjunction with either the JPI window or clipping window libraries.
- Numeric Conversion functions may be used except for those having real numbers as in or out parameters. sscanf and sprintf may be used if the %g|e|f format specifiers are avoided.
- The functions qsort or hqsort may not be used.

- Functions that generate software interrupts (not int86), or use self modifying code may not be used.

Modula-2 Library Considerations

The following groups of functions and variables are not available or are subject to restrictions:

- Command line arguments. ParamStr and ParamCount are not available.
- Post-mortem dump. Not available.
- Program timer. The function Lib.Delay is not available.
- FIO. All functions using file handles may be used on files other than the predefined handles: StandardInput etc.
- Process sub-system. Multi-thread programs may not be used (process.def). Child processes may not be spawned using Lib.Exec etc.
- Floating point. Floating point can be present, but the exception handling mechanism will not be initialized causing library math functions to behave incorrectly. It is therefore not advisable to use floating point functions.
- Memory allocation. Neither the far or near heap functions are available. Sub allocation functions may be used.
- Text Windows. Window functions (window.def) may not be used.
- Graphics. Graphics functions (graph.def) may not be used.
- Environment strings. Not available.
- IO. All console I/O functions can be used, but not in conjunction with the window module.
- Functions that generate software interrupts (not MsDos), or use self-modifying code are not available.

Pascal Library Considerations

The following groups of functions and variables are not available or are subject to restrictions:

- Command line arguments. ParamStr and ParamCount are not available.
- Post-mortem dump. Not available.
- Program timer. The function PasDos.Delay is not available.
- File variables may be used other than the predefined _input and _output.

- Process sub-system. Multi-thread programs may not be used (PasProc.itf). Child processes may not be spawned using PasDos.Exec etc.
- Floating point. Floating point can be present, but the exception handling mechanism will not be initialized causing library math functions to behave incorrectly. It is therefore not advisable to use floating point functions.
- Memory allocation. Neither the far or near heap functions are available. Sub-allocation functions may be used.
- Text Windows. Window functions (PasWin.itf) may not be used.
- Environment strings. Not available.
- TurboCrt. Not Available.
- Functions that generate software interrupts (not MsDos), or use self modifying code are not available.

Extending File Handle Limits

SourceKit Users

The file CoreFile.a may be edited to select the desired number of handles. Change the macro OPEN_MAX to the required total, including predefined handles.

Now remake the libraries or include the file corefile in your project to override the default (this will generate linker warnings).

Header files may be changed to reflect the increased number:

C	change OPEN_MAX, SYS_OPEN, and FOPEN_MAX in stdio.h.
Modula-2	change MaxOpenFiles in FIO.DEF.

Standard Edition Users

Select the appropriate corefile.XXX interface file for your language and edit as instructed in that file.

Extending Threads Limits

To change the number of threads supported by the library the file corelib.inc must be edited, changing the definition of MAXTHREAD to the desired number.

In the appropriate header file for each language (process.h, process.def or process.itf) you must change MaxProcess to reflect this number.

The affected libraries must then be remade.

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

Introduction

In any computer system above a minimum level of complexity, applications do not use the addresses they are given to access memory directly. First they are translated to a new location in system memory. 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

This means 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 speaking, the more complex the system, the more important this distinction becomes.

TopSpeed compilers run on the industry standard 80*x*86 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 80*x*86 range.

The 80x86 Architecture — A Design Compromise

The 80*x*86 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 80*x*86 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 known as *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. If you opt 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. But Pascal, C, Modula-2 and C++ offer extensive and at times exotic pointer facilities, which are almost indispensable to modern object-oriented approaches.

TopSpeed Language Extensions for Memory Models

TopSpeed therefore provides a comprehensive and uniform set of language extensions to deal with the 80*x*86 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 these 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.

Customizing Memory Models

If you wish to go into more detail, TopSpeed gives you full control and complete flexibility. It allows you to define customized memory models — a unique and powerful feature.

Finally, TopSpeed permits *Mixed Model Programming* in which you adopt a standard or custom model that provides the segment organization and default address mechanism, but override the addressing default for selected objects — for example, in a small application with a data space which is small except for one or two large objects that need to be placed in far memory.

The 8086 Architecture

Address architecture is easier to understand by studying how it evolved from the original 8088 and 8086 system. This will provide you with an insight into the reasons for using different memory models, and how to get the most efficient results from TopSpeed compilers.

Segments, Addresses and Registers

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.

Usually, a segment begins on a *paragraph boundary* (an address at a 16-byte boundary) so that a single register can hold its address. This segment address is held in one of the following four segment registers:

CS	**the Code Segment register, addresses the segment containing the program code.**
DS	the Data Segment register, addresses the segment holding the program's global data.
SS	the Stack Segment register, addresses the segment holding the stack.
ES	the Extra Segment register, is used for extra data addressing, for example, for writing to video memory.

If a program has more than one CODE or DATA segment, the contents of the CS and DS registers will change during the execution of the program.

The segment registers do not form the complete picture since you need to address bytes within the segment. Intel termed this form of address the *offset* within the segment. You use the processor's other registers to hold the offset.

Five of the most important registers are:

SP	the Stack Pointer register, points to the top of the stack.
BP	the Base Pointer register, is used to address automatic variables, parameters being passed to called functions, and return addresses from the stack.
SI	the Source Index register, is used for copying strings and pointer indexing.
DI	the Destination Index register, is also used for copying strings and pointer indexing.
IP	the Instruction Pointer register, points to the next instruction within the CODE segment.

The 8086 also has four general-purpose 16-bit registers. You can use each register as two 8-bit registers. For example, you can split the AX register into AH (A High) and AL (A Low) registers.

The four registers are:

AX	often used as an Accumulator.
BX	often used to hold a Base address.
CX	often used for Counting.
DX	often used for Data.

Address Calculation

You usually refer to an absolute address with a segment:offset pair, commonly written with a colon (':') between them.

For example,

```
CS:IP
DS:00A7
DS:BX
```

You calculate an absolute address by multiplying the segment register by 16 (shift left by four) and adding the offset address. For example, given CS=1234 and IP=3456, the absolute address is 1234*16+3456 = 23400.

Real and Protected Modes

The 80286 processor and its successors, the 80386 and 80486 processors, can run in real and protected modes. DOS uses real mode addressing regardless of the machine processor. In real mode, the processor can only address one megabyte of the possible 16M bytes attached to the processor.

OS/2 currently uses 286 protected mode on the 80286, 80386 and 80486 processors. Protected mode addressing still uses two registers to obtain an absolute address with the offset address; this is the same as in real mode. However, in protected mode, the segment registers do not contain the paragraph addresses of segments in memory. Instead, they contain *selectors*, which are indexes into memory tables holding segment descriptors. A *descriptor* contains the 24-bit memory address of the segment, as well as protection flags and segment size.

The addressing mode is transparent to the programmer because you still use a segment:offset pair for addressing. The ability of the processor to protect one program from overwriting other programs requires that pointers cannot point at absolute addresses. If you try to do this, OS/2 gives you a general protection (GP) fault.

Near and Far Pointers

When describing the segment registers, we mentioned that when a program has more than one CODE or DATA segment, the contents of the CS or DS register will change during program execution. This is what using memory models is all about; loading the segment registers as infrequently as possible, while ensuring that they have the correct segment addresses in them at all times.

There are two general types of pointer corresponding to two solutions to this problem. The most general pointer is a four-byte object consisting of a segment:offset pair. This is dereferenced by loading the two segment bytes into a segment register, the two offset bytes into a general register, and then accessing the data via the registers. This kind of pointer is called a *far pointer*.

This is unnecessarily complex if the program has less than 64K bytes of code or data. A two-byte pointer, holding only an offset address, can then be used.

In this case the CS, DS and SS registers need only be set at the beginning of the program, and offset addresses, relative to one of these registers, are used to access code and data. A pointer, therefore, only needs to be an offset address and is called a near pointer.

With far pointers the program can have multiple segments, thereby enlarging it. The segment register contents change according to which data or function is in use.

The overhead of changing the segment registers makes near pointers to data and functions produce quicker and smaller code. Experienced programmers use near pointers as much as possible.

This applies to both real and protected modes but especially in protected mode. The overhead introduced by the protection checking in protected mode makes loading segment registers particularly slow.

Using Memory Models

TopSpeed offers several methods of controlling pointer size, but the easiest and safest is to use the correct memory model. A memory model sets defaults for address handling which apply uniformly to all pointers, minimizing the risk of error arising from different pointer representations. A further advantage of using a memory model is that it cuts down the number of non-standard constructs in your program, making it more portable.

TopSpeed's standard memory models are described below. You should use the smallest memory model that suits the size of your program. Unfortunately, changing models in mid-development can cause many problems, so it's best to err on the large side.

Linking with Memory Models: Segment Registers, Classes and Groups

A memory model serves many useful purposes. First of all, it decides when to use near pointers and when to use far ones. It also tells the linker how to lay out your program and data in your address space.

The underlying aim is to reset segment registers as infrequently as possible. An ideal program would set up its CS, DS and SS registers once only, on entry, and then reference everything relative to one or other of them. Then it would only need to load offset registers to calculate addresses. One of the linker's jobs is to organize your memory so that your program can get as close as possible to this theoretical ideal.

TopSpeed produces object files in a format which assists the linker to achieve this. It communicates with the linker using the object-language constructs *segment*, *group* and *class* developed by Intel, the designers of the 80*x*86 family. Group and class classifications help the linker 'bunch' items in chunks of less than 64K which can all be addressed using a single segment register. The system can be summarized as follows:

- An item is a fundamental, indivisible unit of code or data, sometimes called a logical segment. In this manual we will use the more current term, object. Some confusion with object code may result; where the context doesn't resolve it we talk of data objects.
- A segment is a collection of objects whose total size is less than 64K bytes; a segment has a name. Any object belongs to only one segment.
- A group is a collection of segments whose total size is less than 64K bytes; a group also has a name. Any segment belongs to at most one group, but need not belong to any.
- Segments may be qualified by a named class. Classes are used to specify preference when arranging the objects: objects with the same class name are placed adjacent to each other; within a class, objects from the same segment are placed adjacent to each other.

Classes take precedence, so if two objects have the same segment name but different class names, they are not even considered to belong to the same segment.

If these conventions are observed, the linker ensures that you can address all the objects in a given segment or group without having to reload a segment register.

Default Segment Register Assignments

We are nearly ready to tackle the standard memory models. It remains only to explain the default segment register assignments.

Intel's design imposes a standard segment register usage which is nearly always adhered to.

- The program or code is always addressed relative to CS, the CODE segment register. 'Near' jumps (within a single CODE segment) do not change it; 'far' jumps (between segments) change CS to point at the beginning of the segment being jumped into. All CODE segments are of class CODE.
- All global objects which do not require a separate segment of their own, are put in a single group called DGROUP. The DATA segment register DS points at this on entry to the program, and in all models except extra large, remains pointing there.
- The stack is always addressed relative to SS, the stack segment register. It always points to the start of the stack. In the smaller memory models the stack is in DGROUP; if it is too big for this, it has a separate segment.

The Standard Memory Models

TopSpeed compilers provide six standard memory models. The *small memory 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	64Kb	64Kb	1Mb*	1Mb*	1Mb*
Data size	64Kb	1Mb*	64Kb	1Mb*	1Mb*
Code pointers	near	near	far	far	far
Data pointers	near	far	near	far	far

* 16Mb 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 reentrant library.
- The overlay model is the multi-thread model plus segment-based overlaying.
- The dynalink model is the multi-thread model plus dynamic

linking. Under DOS it also includes segment-based overlays.

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. For further information on the standard JPI memory models please refer to the "*TopSpeed Developer's Guide*". The information in this manual includes a discussion of mixed-model programming.

Small Memory Model

The *small* memory model is the most efficient, which is why it 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 (see '*Mixed Model Programming*', later on in this chapter).The following diagram illustrates the small memory model:

```
     Low address
   _TEXT class CODE
  up to 64Kb of code = CS
_CONST class DATA
constant data
DGROUP  = DS, SS
_DATA class DATA
initialized data
DGROUP
_BSS class DATA
uninitialized data
DGROUP
HEAP
STACK = SP grows towards far heap
  FAR HEAP
  Rest of available memory

  High address
```

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.

- No single data object may exceed 64Kb.
- No single source file may produce more than 64Kb of code.
- The sum of all data objects below the threshold must not exceed 64Kb.

The following diagram illustrates the large memory model:

```
    Low address
  <file-name>_TEXT class CODE
up to 64Kb code per file
= CS points to the 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 points to the active data segment
_DATA class DATA
initialized data
DGROUP
_BSS class DATA
uninitialized data
DGROUP
HEAP  Only created if used
STACK = SS
= SP grows towards heap
  FAR HEAP
  High address
```

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 in 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 together 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:

```
    Low address
  <file-name>_TEXT class CODE
up to 64Kb code per file
= CS points at active code segment.
_CONST class DATA
constant data
DGROUP  = DS,SS
_DATA class DATA
initialized data
DGROUP
_BSS class DATA
uninitialized data
DGROUP
HEAP
STACK = SP, grows towards HEAP
  FAR HEAP
Up to the rest of available memory
  High address
```

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 the 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 reentrant library. You should use the multi-thread model when writing multi-threading programs using 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.

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 one of the following:

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.

Choosing How to Pass Parameters

By default, TopSpeed uses a register-based calling convention, where function parameters are passed in registers wherever possible, in order to speed up program execution. If you require the more usual convention of passing all parameters on the stack (for example, to interface to some external library), you should specify this in the #model command in your project file. This can be achieved using the Project Project options (J) Calling convention menu command.

Changing the calling convention in this way will cause stack-based parameters to be used throughout your program, and will require an alternative set of libraries to be linked. The TopSpeed system is supplied with stack-based libraries for the *large* memory model only, but if you have purchased the library source code, a set of stack-based libraries may be constructed for any memory model.

An alternative approach is to use the stack-based calling convention only for certain procedures, by means of pragmas or special keywords at the declaration of those procedures.

Mixed Model Programming

Though it is easiest to work with the most appropriate standard memory model, this may not give the most efficient code. For example, if a program only needs one CODE and one DATA segment, then the small memory model should be chosen; but if this program wants access to an absolute address outside its own segment it will need a far pointer.

This problem could be solved by using the compact model. But it brings an unnecessary overhead, since it will use far pointers for all data objects even when they could be accessed without resetting any segment registers. This is where *Mixed Model Programming* becomes useful. It lets you use a far pointer to access the far segment, while using near pointers for all other data.

Mixed model programming is most often used with the *small* memory model, providing near pointers as default. The compiler is then told to use far pointers for specified data objects.

There are two ways to do this. You can use non-standard pointer declarations which explicitly designate a given pointer as near, far or huge; or you can use pragmas which override the model's defaults for a specific part of the program.

Near and Far Pointer Declarations

The syntax for these is slightly different for C/C++ and for Pascal/Modula-2. However the underlying concept, and the 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 they 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 two-byte near pointer, whatever memory model the program is compiled with. z will always be a four-byte 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 a 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 current 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 the "*TopSpeed Developer's Guide*" 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 be 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 type far pointer, 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 the "*TopSpeed Developer's Guide*" 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.

The most important arises when you add or subtract from a pointer. You have to be careful that its value does not exceed 65535 or go lower than 0. This causes *segment wrap-around* and 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 3 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 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 *) halloc (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 (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;

  x, y : CARDINAL;

BEGIN
  FOR y := 1 TO 25 DO
    FOR x := 1 TO 80 DO
      WITH ScrMem[y,x] DO
        Chr := '*';
        Atr := 87H;
      END;
    END;
  END;
END MonoMem.
```

An alternative way of writing the pointer declaration in C++ is to use a relative pointer, which forces a pointer to use a given variable to preload the segment register. For example:

```
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 de-reference ptr, so seg can be updated if necessary.

Pragmas

Mixed model programming has traditionally been achieved using the keywords near and far. These keywords are not in the ANSI 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 the "*TopSpeed Developer's Guide*".

You can use the data(seg_name) 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 declare data pointers to be near or far pointer 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 call is 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 in this example 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. 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.

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 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 (* 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(near_call) pragma affects both the type of call used for the function *and* the size of any parameters which are themselves pointers to functions. 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 (*FARFNCPTR) (void);
#pragma call(near_call => on)
int NearFunc(FARFNCPTR FParm);
#pragma restore
```

Changing the Global Threshold

The *compact*, *large*, *extra large* and *multi-thread* memory models place objects larger than the threshold in a separate segment. The default threshold value is 32Kb. However, this value won't suit you if, for example, your program has three data objects of size 30Kb. These data objects will all be placed in a single segment, which will then exceed 64Kb, causing an error at link time. To overcome this, you need to alter the threshold size to below 30Kb. The threshold is controlled using the data(threshold) pragma. This pragma may be specified in a project file to change the threshold globally in a project, or it may be specified in a source file so that only certain data objects are affected.

APPENDIX B
MODULE DEFINITION FILE SYNTAX

A module definition file describes the name, attributes, exports, and other characteristics of an application or library for DOS (using the TopSpeed Overlay System), OS/2 or Microsoft Windows. This file is required for Windows applications and libraries, and is also required for overlay programs and dynamic-link libraries that run under DOS and OS/2.

Syntax of the Module Definition File

A module definition file contains one or more statements. Each statement defines an attribute of the executable file, such as its module name, the attributes of program segments, and the numbers and names of exported symbols. The statements and the attributes they define are listed below:

Statement	Attribute
NAME	Names application
LIBRARY	Names dynamic-link library
CODE	Gives default attributes for CODE segments
DATA	Gives default attributes for DATA segments
SEGMENTS	Gives attributes for specific segments
STACKSIZE	Specifies local stack size in bytes
EXPORTS	Defines exported functions
HEAPSIZE	Specifies local heap size
PROTMODE	Flags file as protected mode only
REALMODE	Flags file as real mode only
EXETYPE	Identifies operating system

The following rules govern the use of these statements in a module definition file:

- If you use either a NAME or a LIBRARY statement, it must precede all other statements in the module definition file.
- You can include source-level comments in the module definition file, by beginning a line with a semicolon(;). The utilities ignore each such comment line.
- Module definition keywords (such as NAME, LIBRARY, and SEGMENTS) must be entered in uppercase letters.

The following example gives module definitions for a dynamic-link library:

```
LIBRARY MyDLL
; Sample export file

EXPORTS
    Func1   @1
    Var1    @2
    Func2   @3
    Func3   @4
    Func4   @5
```

The NAME Statement

The NAME statement identifies the file as an executable application (rather than a DLL) and optionally defines the name and application type.

Syntax

```
NAME [appname][apptype]
```

If appname is given, it becomes the name of the application as it is known by the operating system. If no appname is given, the name of the executable file - with the extension removed - becomes the name of the application.

The apptype field is used to control the program's behavior under Windows and Presentation Manager (PM). This information is kept in the executable-file header. You do not need to use this field unless you may be using your application in a Windows or PM environment. The apptype field may have one of the following values:

Keyword	Meaning
WINDOWAPI	Windows or PM application. The application uses the API provided by Windows or PM and must be executed in the Windows or PM environment.
WINDOWCOMPAT	

Window-compatible application. The application can run full-screen or inside a Windows or PM window.

NOTWINDOWCOMPAT
Application can only run full-screen.

If the NAME statement is included in the module-definition file, then the LIBRARY statement cannot appear.

If neither a NAME statement nor a LIBRARY statement appears in a module-definition file, NAME is assumed.

The following example assigns the name wdemo to the application being defined:

```
NAME wdemo  WINDOWAPI
```

The LIBRARY Statement

The LIBRARY statement identifies the file as a dynamic-link library. The name of the library, and the type of library module initialization required, may also be specified.

Syntax

```
LIBRARY [libraryname][initialization]
```

If libraryname is specified, it becomes the name of the library as it is known by the operating system. This name can be any valid file name. If no libraryname is given, the name of the executable file - with the extension removed - becomes the name of the library.

The initialization field is optional and can have one of the two values listed below. If neither is given, then the initialization default is INITINSTANCE.

Keyword	Meaning
INITGLOBAL	The library-initialization routine is called only when the library module is initially loaded into memory.
INITINSTANCE	The library-initialization routine is called each time a new process gains access to the library.

If the LIBRARY statement is included in a module definition file, then the NAME statement cannot appear.

The following example assigns the name mydll to the dynamic-link module being defined, and specifies that library initialization is performed each time a new process gains access to myDLL:

```
LIBRARY myDLL INITINSTANCE
```

The CODE Statement

The CODE statement defines the default attributes for CODE segments within the application or library.

Syntax:

```
CODE [attribute...]
```

Each attribute specified must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are presented below, along with legal values. In each case, the default value is listed last. The last three fields have no effect on OS/2 code segments and are included for use with Microsoft Windows.

Field	Values
load	PRELOAD, LOADONCALL
executeonly	EXECUTEONLY, EXECUTEREAD
iopl	IOPL, NOIOPL
conforming	CONFORMING, NONCONFORMING
shared	SHARED, NONSHARED
moveable	MOVEABLE, FIXED
discard	NONDISCARDABLE, DISCARDABLE

The *load* field determines when a code segment is to be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded automatically, at the beginning of the program, and, when using the TopSpeed Overlay System, will not be swapped out.
LOADONCALL	The segment is not loaded until accessed (default).

The *executeonly* field determines whether a code segment can be read as well as executed. This field contains one of the following keywords:

Keyword	Meaning
EXECUTEONLY	The segment can only be executed.
EXECUTEREAD	The segment can be both executed and read (default).

The *iopl* field determines whether or not a segment has I/O privilege (that is, whether it can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The CODE segment has I/O privilege.
NOIOPL	The CODE segment does not have I/O privilege (de-

fault).

The *conforming* field specifies whether or not a code segment is a 286 conforming segment. The concept of a conforming segment deals with privilege level (the range of instructions that the process can execute) and is relevant only to users writing device drivers and system-level code. A conforming segment can be called from either Ring 2 or Ring 3, and the segment executes at the caller's privilege level. This field contains one of the following keywords: CONFORMING or NONCONFORMING (the default).

The *shared* field determines whether all instances of the program can share a given code segment. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all code segments are shared. The *shared* field contains one of the following keywords: SHARED or NONSHARED (the default).

The *moveable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *moveable* field contains one of the following keywords: MOVEABLE or FIXED (the default for Windows).

The *discard* field determines whether a segment can be swapped out to disk by the operating system when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments can be swapped as needed. The *discard* field contains one of the following keywords: DISCARDABLE or NONDISCARDABLE (the default for Windows).

The following example sets defaults for the module's code segments, so that they are not loaded until accessed and so that they have I/O hardware privilege:

```
CODE LOADONCALL IOPL
```

The DATA Statement

The DATA statement defines the default attributes for the DATA segments within the application or library.

Syntax:

```
DATA [attribute...]
```

Each attribute must correspond to one of the following attribute fields. Each field can appear at most once, and order is not significant. The attribute fields are listed below, along with the legal values for each field. In each case, the default value is listed last. The last two fields have no effect on OS/2 DATA segments, but are included for use with Microsoft Windows.

Field	Values
load	PRELOAD, LOADONCALL
readonly	READONLY, READWRITE
instance	NONE, SINGLE, MULTIPLE
shared	SHARED, NONSHARED
moveable	MOVEABLE, FIXED
discard	DISCARDABLE, NONDISCARDABLE

The *load* field determines when a segment will be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded when the program begins execution and, when using the TopSpeed Overlay System, will not be swapped out.
LOADONCALL	The segment is not loaded until it is accessed (default).

The *readonly* field determines the access rights to a DATA segment. This field contains one of the following keywords:

Keyword	Meaning
READONLY	The segment can only be read.
READWRITE	The segment can be both read and written to (default).

The *instance* field affects the sharing attributes of the automatic DATA segment, which is the physical segment represented by the group name DGROUP. (This segment group makes up the physical segment which contains the local stack and heap of the application.) This field contains one of the following keywords:

Keyword	Meaning
NONE	No automatic DATA segment is created.
SINGLE	A single automatic DATA segment is shared by all instances of the module. In this case, the module is said to have “solo” data. This keyword is the default for dynamic-link libraries.
MULTIPLE	The automatic DATA segment is copied for each instance of the module. In this case, the module is said to have “instance” data. This keyword is the default for applications.

The *shared* field determines whether all instances of the program can share a READWRITE DATA segment.

The *moveable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-

mode Windows. Under OS/2, all segments are moveable. This field contains one of the following keywords: MOVABLE or FIXED (the default for Windows).

The optional *discard* field determines whether a segment can be swapped out to disk by the operating system, when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows and the TopSpeed Overlay System. Under OS/2 systems, all segments can be swapped as needed. This field contains one of the following keywords: DISCARDABLE or NONDISCARDABLE (the default for Windows).

Warning:
Care should be taken not to specify contradictory segment attributes.

The following example defines the default attributes for DATA segments so that they are loaded only when accessed, and cannot be shared by more than one copy of the program:

```
DATA LOADONCALL NONSHARED
```

By default, the DATA segment can be read and written, and the automatic DATA segment is copied for each instance of the module.

The SEGMENTS Statement

The SEGMENTS statement defines the attributes of one or more individual segments in the application or library on a segment-by-segment basis. The attributes specified by this statement override defaults set in CODE and DATA statements.

Syntax:

SEGMENTS
    segmentdefinitions

The SEGMENTS keyword marks the beginning of the segment definitions. This keyword can be followed by one or more segment definitions, each on a separate line. The syntax for each segment definition is as follows:

```
segname [CLASS'classname'][attribute...]
```

Each segment definition begins with a *segment name*, which can be placed in optional single quotation marks (‘). The quotation marks are required if the *segment name* conflicts with a module definition keyword, such as CODE or DATA.

The CLASS keyword specifies the class of the segment. The single quotation marks (‘) are required around classname. If you do not use the CLASS argument, the class is assumed to be CODE.

Each attribute must correspond to one of the attribute fields described for the CODE and DATA statements above. Each field can appear at most once, and order is not significant.

The following example specifies segments named pr1_TEXT, pr2_TEXT and pr3_TEXT. Each segment is given different attributes.

```
SEGMENTS
  pr1_TEXT IOPL
  pr2_TEXT EXECUTEONLY PRELOAD
  pr3_TEXT LOADONCALL READONLY
```

The STACKSIZE Statement

The STACKSIZE statement specifies the stack size for a Windows or Presentation Manager program only.

Syntax:

```
STACKSIZE number
```

The number must be an integer. The number is considered to be in decimal format by default, but you can use C notation to specify hexadecimal or octal.

The following example allocates 4096 bytes of local stack space:

```
STACKSIZE 4096
```

The HEAPSIZE Statement

The HEAPSIZE statement defines the size of the application's local heap, in bytes. This value affects the size of the automatic DATA segment.

Syntax:

```
HEAPSIZE bytes
```

The bytes field is an integer number, which is considered decimal by default. However, hexadecimal and octal numbers can be entered by using C notation. For example:

```
HEAPSIZE 4000
```

The EXPORTS Statement

The EXPORTS statement defines the names and attributes of the functions exported to other modules, and of the functions that run with I/O privilege. The term "export" refers to the process of making a function available to other run-time modules. By default, functions are hidden from other modules at run time.

Syntax:

EXPORTS
    exportdefinitions

The EXPORTS keyword marks the beginning of the export definitions. It may be followed by up to 3072 export definitions, each on a separate line. You need to give an export definition for each dynamic-link routine that you want to make available to other modules. The syntax for an export definition is as follows:

```
entryname [pwords] @ Number|? [NODATA]
```

The entryname specification defines the function name as it is known to other modules.

The ordinal field (introduced by the @ character) defines the function's ordinal position within the module-definition table. The numbers must either be in sequence or use the ? character.

The pwords field specifies the total size of the function's parameters, as measured in words (the total number of bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults the pwords field to determine how many words to copy from the caller's stack to the I/O-privileged function's stack.

The optional keyword NODATA is ignored by OS/2, but is provided for use by real-mode Windows.

The EXPORTS statement is meaningful for functions within dynamic-link libraries, functions which execute with I/O privilege, and call back functions in Windows programs.

For example:

```
EXPORTS
  Func1      @?
  Func2    @?
  CharTest   @?
```

The PROTMODE Statement

This statement is ignored.

The REALMODE Statement

This statement is ignored.

The EXETYPE Statement

The EXETYPE statement specifies in which operating system the application (or dynamic-link library) is to run. This statement is optional and provides an additional degree of protection against the program being run in an incorrect operating system.

Syntax:

```
EXETYPE [OS/2 | WINDOWS | DOS4]
```

The effect of EXETYPE is simply to set bits in the header which identify operating system type. The operating system loaders may or may not check these bits. When writing programs for Microsoft Windows, EXETYPE WINDOWS must be specified.

APPENDIX C
DOS FUNCTION CALLS

This appendix lists the DOS function call numbers and names, and gives a brief description of their parameters and return values as used by the Watch program (see Chapter : '*Watch*'). It is not a tutorial in the use of the function calls - there are many excellent technical reference books giving a complete background to these functions.

The functions are listed in order of their function number. This is the function code that is loaded into the AH register before the int 21H instruction is issued by the calling program. The exceptions are the NETBIOS calls 5E??H and 5F??H; they have the full contents of the AX register listed. For further details of how DOS function calls are used in MS-DOS programs, see Chapter : '*Watch*'.

00h : PROGRAM TERMINATE

INPUT `CS` contains the segment of PSP to terminate.
OUTPUT None.

01h : WAIT FOR KEYBOARD INPUT

INPUT None.

OUTPUT AL contains the character from stdin.

02h : DISPLAY OUTPUT

INPUT `DL` contains the character to write to `stdout`.
OUTPUT None.

03h : AUXILIARY INPUT

INPUT None.
OUTPUT `AL` contains the character from `stdaux`.

04h : AUXILIARY OUTPUT

INPUT `DL` contains the character to write to `stdaux`.
OUTPUT None.

05h : PRINTER OUTPUT

INPUT `DL` contains character to write to `stdprn`.
OUTPUT None.

06h : DIRECT CONSOLE I/O

INPUT If `DL` is not `FFh` then the `DL` is treated as a character and written directly to the console, otherwise direct console input is made from the console.

OUTPUT None for direct console output. For direct console input the Zero Flag (ZF) indicates the input status:
= 1 no character available
= 0 AL contains character from stdin

07h : DIRECT CONSOLE I/O; NO ECHO

INPUT None.
OUTPUT `AL` contains the character, which has not been echoed.

08h : CONSOLE INPUT; NO ECHO

INPUT None.
OUTPUT `AL` contains the next character from `stdin`.

09h : DISPLAY STRING

INPUT `DS:DX` contains the address of a string which is terminated with a `$` (which is not displayed on the console).
OUTPUT None.

0Ah : BUFFERED KEYBOARD INPUT

INPUT DS:DX is the address of an input buffer:
1st byte: length of buffer.
Remainder: undefined.

The buffer should have enough room to hold the maximum expected message plus 2 bytes. The carriage return is not stored in the buffer.

OUTPUT DS:DX still contains the address of the input buffer, but the buffer now contains:
1st byte: length of buffer
2nd byte: number of bytes entered
Remainder: the bytes entered.

0Bh : STDIN STATUS

INPUT None.

OUTPUT AL indicates the status of stdin:
= FFh character available
!=FFh no character available.

0Ch : CLEAR KEYBOARD; INVOKE FUNCTION

INPUT — AL contains a function number (01h, 06h, 07h, 08h, or 0Ah) to be called after the keyboard buffer has been cleared, other registers as defined for that function number.

OUTPUT — As defined for the function number that it was called with.

0Dh : DISK RESET

INPUT — None.

OUTPUT — None.

0Eh : SELECT DISK DRIVE

INPUT — DL: drive to select (A = 0, B = 1, etc.).

OUTPUT — AL contains value of LASTDRIVE - set in CONFIG.SYS (default E).

0Fh : (FCB) OPEN FILE

INPUT — DS:DX contains address of unopened FCB containing the filename.

OUTPUT — Status is returned in AL:
= 00h file was opened OK
= FFh an error occurred.

10h : (FCB) CLOSE FILE

INPUT — DS:DX contains the address of an open FCB.

OUTPUT — Status is returned in AL:
= 00h file closed
= FFh file not closed.

11h : (FCB) FIND FIRST

INPUT — DS:DX contains the address of an FCB which is not used for an open file. The file name can contain either a full file name or can contain the ? wild card.

OUTPUT — Status is returned in AL:
= 00h match was found
= FFh no match was found.

12h : (FCB) FIND NEXT

INPUT — DS:DX contains the address of an FCB which has previously been used by a function 11h call (see above).

OUTPUT — Status is returned in AL:
00h match was found
FFh no match found.

13h : (FCB) DELETE FILE

INPUT `DS:DX` contains the address of an FCB which is not being used by an open file. The file name should be set to the file the program wishes to delete.

OUTPUT Status is returned in AL:
= 00h file was deleted
= FFh file not deleted.

14h : (FCB) SEQUENTIAL READ

INPUT `DS:DX` contains the address of an FCB for a file which has been opened.

OUTPUT Status is returned in AL:
= 0 read successful
= 1 file already at EOF
= 2 DTA too small
= 3 partial read; at EOF.

15h : (FCB) SEQUENTIAL WRITE

INPUT `DS:DX` contains the address of an FCB for a file which has been opened.

OUTPUT Status is returned in AL:
= 0 write successful
= 1 disk full
= 2 DTA too small.

16h : (FCB) CREATE FILE

INPUT `DS:DX` contains the address of an FCB which is not in use by an open file. The file name should be set to the file the program wishes to create.

OUTPUT Status is returned in AL:
= 00h file was created
= FFh file not created.

17h : (FCB) RENAME FILE

INPUT `DS:DX` contains the address of FCB suitably modified.

OUTPUT Status is returned in AL:
= 00h file was renamed
= FFh file not renamed.

19h : GET CURRENT DISK

INPUT None.
OUTPUT `AL` contains a number indicating the current drive. (A: = 0, B: = 1, etc.).

1Ah : SET DISK TRANSFER AREA (DTA)

INPUT `DS:DX` address of the new DTA.
OUTPUT None.

1Bh : GET ALLOCATION DATA

INPUT None.

OUTPUT Default drive data:
DS:BX address of media descriptor byte.
DX number of clusters
CX bytes per sector
AL sectors per cluster.

1Ch : GET ALLOCATION DATA FOR DISK

INPUT `DL` contains a number of the drive.
(current drive = 0, A = 1, B = 1, etc.).

OUTPUT Specified drive data:
DS:BX address of media descriptor byte
DX number of clusters
CX bytes per sector
AL sectors per cluster.

21h : (FCB) RANDOM READ

INPUT `DS:DX` contains the address of an FCB for a file which has been opened

OUTPUT Status is returned in AL:
= 0 read successful
= 1 file already at EOF
= 2 DTA too small
= 3 partial read; at EOF.

22h : (FCB) RANDOM WRITE

INPUT `DS:DX` contains the address of an FCB for a file which has been opened.

OUTPUT Status is returned in AL:
= 0 write successful
= 1 disk full
= 2 DTA too small.

23h : (FCB) GET FILE SIZE

INPUT `DS:DX` contains the address of an FCB which is not being used by an open file. The file name should be set to the file of interest.

OUTPUT: Status is returned in AL:
= 00h OK
= FFh file not found

If the call is successful the *Random Record Field* of the FCB contains the size of the file.

24h : (FCB) SET RELATIVE RECORD FIELD

INPUT `DS:DX` contains the address of an FCB for a file which has been opened.

OUTPUT None.

25h : SET INTERRUPT VECTOR

INPUT `AL` contains the interrupt number
`DS:DX` contains the address of the new interrupt handler.

OUTPUT None.

26h : CREATE PROGRAM SEGMENT PREFIX

INPUT `DX` segment address for the new PSP area (i.e. the new PSP is at `DX:0000`).

OUTPUT None.

27h : (FCB) RANDOM BLOCK READ

INPUT `DS:DX` contains the address of an FCB for a file which has been successfully opened.

OUTPUT Status is returned in AL:
= 0 read successfu.
= 1 file already at EOF
= 2 DTA too small
= 3 partial read; at EOF.

CX is set to the number of records read.

28h : (FCB) RANDOM BLOCK WRITE

INPUT `DS:DX` contains the address of an FCB for a file which has been successfully opened
`CX` should be set to the number of records to write.

OUTPUT Status is returned in AL:
= 0 write successful
= 1 disk full
= 2 DTA too small

CX is set to the number of records actually written.

29h : (FCB) PARSE FILENAME

INPUT DS:SI contains the address of command line buffer.
AL contains the parsing instructions which are bit-significant:
Bit 7 -Ì
Bit 6 û- Reserved
Bit 5 °
Bit 4 -ì
Bit 3 Set extension
Bit 2 Set file name
Bit 1 Set Drive ID byte
Bit 0 Strip off leading spaces

OUTPUT ES:DI contains the address of newly-created FCB.
DS:SI contains the address of first character after the parsed filename.
Status is returned in AL:
= 00h Full file name found
= 01h Wildcards (‘*’ or ‘?’) found
= FFh Drive letter invalid

2Ah : GET DATE

INPUT None.
OUTPUT
`DH` month (Jan = 1, Dec = 12).
`DL` day (1 - 31).
`CX` year.
`AL` day of week (Sunday = 0).

2Bh : SET DATE

INPUT
`DH` month (Jan = 1, Dec = 12).
`DL` day (1-31).
`CX` year.

OUTPUT Status returned in AL:
= 00h date was set
= FFh date not valid (not set)

2Ch : GET TIME

INPUT None.
OUTPUT
- CH hour (0-23).
- CL minutes (0-59).
- DH seconds (0-59).
- DL hundredths (0-99).

2Dh : SET TIME

INPUT
- CH hour (0-23).
- CL minutes (0-59).
- DH seconds (0-59).
- DL hundredths (0-99).

OUTPUT Status in AL:
- = 00h time was set
- = FFh time not valid (not set)

2Eh : SET VERIFY ON/OFF

INPUT Parameter in AL:
- = 0 set verify off
- = 1 set verify on

OUTPUT None.

2Fh : GET DISK TRANSFER AREA

INPUT None.
OUTPUT ES:BX contains the address of the current DTA.

30h : GET DOS VERSION

INPUT None.
OUTPUT
- AL major version number.
- AH minor version number.

31h : TERMINATE, BUT STAY RESIDENT

INPUT
- AL return code.
- DX number of paragraphs to retain.

OUTPUT None.

33h : SET/GET CTRL-BREAK

INPUT
- AL = 0 get Ctrl-Brk setting.
- AL = 1 set Ctrl-Brk by value in DL.
- AL = 0 set BREAK off.
- AL = 1 set BREAK on.

OUTPUT New/Current Value of Ctrl-Brk setting is returned in DL.
- = 0 BREAK is off
- = 1 BREAK is on

34h : GET IN-DOS FLAG

INPUT None.
OUTPUT `ES:BX` contains the address of the `In-DOS` flag.

35h : GET INTERRUPT VECTOR

INPUT `AL` interrupt number.
OUTPUT `ES:BX` current address of interrupt handler.

36h : GET DISK FREE SPACE

INPUT `DL` drive number.
(current drive = 0, A = 1, B = 2, etc.).
OUTPUT If `AX` contains `FFFFh` the drive number was invalid.
Otherwise:
AX sectors per cluster
DX total clusters
BX free clusters
CX bytes per sector

38h : SET COUNTRY INFORMATION

INPUT DX must be FFFFh.
The country code is passed in AL or BX, depending on the value of AL.
= FFh country code in BX
!= FFh country code in AL
OUTPUT If Carry flag is set them `AX` contains an error code.

38h : GET COUNTRY INFORMATION

INPUT DS:DX contains the address of a buffer to hold the country information. The country information retrieved depends on the settings of AL.
= 00h get current country
= FFh get for country code in BX
!= FFh get for country code in AL
OUTPUT If Carry flag is set then `AX` contains an error code, otherwise `BX` contains the country code and the buffer `DS:DX` is updated with the country-specific information.

39h : CREATE SUB-DIRECTORY

INPUT `DS:DX` contains the address of an `ASCIIZ` sub-directory name string.
OUTPUT If Carry flag is set then `AX` contains an error code.

3Ah : REMOVE SUB-DIRECTORY

INPUT `DS:DX` contains the address of an `ASCIIZ` sub-directory name string.
OUTPUT If Carry flag is set then `AX` contains an error code.

3Bh : CHANGE SUB-DIRECTORY

INPUT `DS:DX` contains the address of an `ASCIIZ` sub-directory name string.
OUTPUT If Carry flag is set then `AX` contains an error code.

3Ch : CREATE FILE

INPUT `DS:DX` contains the address of an `ASCIIZ` file name string.
OUTPUT If Carry flag is set then `AX` contains an error code, otherwise `AX` contains a file handle for the open, newly-created, file.

3Dh : OPEN FILE

INPUT `DS:DX` contains address of an `ASCIIZ` file name string
`AL` contains the access/sharing mode bit mask.

OUTPUT If Carry flag is set then AX contains an error code, otherwise AX contains a file handle for the open file.

3Eh : CLOSE FILE

INPUT `BX` contains the file handle to close.
OUTPUT If Carry flag is set then `AX` contains the error code.

3Fh : READ FILE

INPUT `BX` file handle.
`CX` number of bytes to read.
`DS:DX` address of I/O buffer.
OUTPUT If Carry flag is set then `AX` contains an error code, otherwise `AX` contains the actual bytes read.

40h : WRITE FILE

INPUT `BX` file handle.
`CX` number of bytes to write.
`DS:DX` address of I/O buffer.
OUTPUT If Carry flag is set then `AX` contains an error code, otherwise `AX` contains the actual bytes written.

41h : DELETE FILE

INPUT — DS:DX contains the address of an ASCIIZ file name string.

OUTPUT — If Carry flag is set then AX contains an error code.

42h : SEEK

INPUT — AL contains a value indicating the origin to use when performing the seek.

= 0 from beginning of file
= 1 from current position
= 2 from end of file

CX:DX contains a long int indicating the number of bytes to move.
BX contains the file handle.

OUTPUT — If Carry flag is set then AX contains an error code, otherwise DX:AX contains a long int indicating the new file position.

43h : GET/SET FILE MODE

INPUT — DS:DX contains the address of an ASCIIZ file name string. AL determines the sub-function performed:

= 0 get file mode
= 1 set file mode

If the file mode is to be set then CX contains the new file mode.

OUTPUT — If Carry flag is set then AX contains an error code, otherwise CX contains the new/current file mode bits.

44h : I/O CONTROL

INPUT AL contains the sub-function code:

00h	get device information
01h	set device information
02h	read character device
03h	write to char device
04h	read block device
05h	write to block device
06h	get input status
07h	get output status
08h	is device changeable?
09h	is network device?
0Ah	is network handle?
0Bh	set sharing retry/count
0Ch	character generic IOCTL
0Dh	block generic IOCTL
0Eh	get logical drive
0Fh	set logical drive

The other registers must be set as necessary for the sub-function.

OUTPUT Depends on the sub-function.

45h : DUPLICATE HANDLE

INPUT `BX` existing file handle.

OUTPUT If Carry flag is set then `AX` contains an error code, otherwise `AX` contains the duplicate handle.

46h : FORCE DUPLICATE HANDLE

INPUT `BX` existing file handle.
`CX` desired duplicate handle.

OUTPUT If Carry flag is set then `AX` contains an error code.

47h : GET CURRENT SUB-DIRECTORY

INPUT `DS:SI` contains the address of 64-byte buffer to hold the directory path. `DL` contains the drive number (current drive = 0, A = 1, B = 2, etc.).

OUTPUT If Carry flag is set then `AX` contains an error code, otherwise the buffer pointed to by `DS:SI` contains full pathname.

48h : ALLOCATE MEMORY

INPUT `BX` contains the number paragraphs to allocate.

OUTPUT If Carry flag is set then `AX` contains an error code and `BX` contains the size of largest free memory block. Otherwise, `AX` is the segment address of the allocated block (i.e. `AX:0000` is the start of the block).

49h : FREE ALLOCATED MEMORY

INPUT ES contains the segment address of the block to be freed (i.e. ES:0000 is the start of the block).

OUTPUT If Carry flag is set then AX contains an error code.

4Ah : MODIFY ALLOCATED MEMORY

INPUT ES contains the segment address of the block to be modified (i.e. ES:0000 is the start of the block) and BX contains the new size in paragraphs.

OUTPUT If Carry flag is set then AX contains an error code otherwise BX contains the maximum size for the block.

4Bh : LOAD/EXECUTE PROGRAM

INPUT DS:DX contains the address of an ASCIIZ file name string for the executable file. AL contains the sub-function code.

= 0 load and execute
= 3 load as overlay

ES:BX contains the address of the parameter block which has the structure:

For AL = 0:

WORD	segment of environment
DWORD	address of command line
DWORD	address of 1st FCB
DWORD	address of 2nd FCB

For AL = 3:

WORD	segment at which to load
WORD	relocation factor

OUTPUT If Carry flag is set then AX contains an error code
NOTE: On return from LOAD AND EXECUTE all registers are destroyed.

4Ch : TERMINATE PROGRAM

INPUT AL contains return code (ERRORLEVEL).

OUTPUT None.

4Dh : GET TERMINATION CODE

INPUT None.

OUTPUT AH contains a number indicating the type of the termination:

= 0 normal
= 1 Ctrl-Break
= 2 Critical error
= 3 TSR via function 31h

AL contains the return code of the process (ERRORLEVEL).

4Eh : FIND FIRST MATCHING FILE

INPUT `DS:DX` contains the address of an `ASCIIZ` file name string, possibly containing '`*`' and/or '`?`' wildcard chars.
`CX` contains the file attribute word to use in the search.

OUTPUT If Carry flag is set then AX contains an error code, otherwise the DTA is set with the return result:

21 bytes	reserved
1 byte	file attributes
1 WORD	file's time
1 WORD	file's date
1 DWORD	file's size
13 bytes	filename

4Fh : FIND NEXT FILE

INPUT None, but DTA must still be set from the previous call to `FIND FIRST` or `FIND NEXT`.

OUTPUT Same as for `FIND FIRST`.

52h : GET DOS VARIABLES

INPUT None.

OUTPUT `ES:BX` contains address of pointers to DOS variables.

54h : GET VERIFY STATUS

INPUT None.

OUTPUT Setting is in AL:

= 0	VERIFY is OFF
= 1	VERIFY is ON

56h : RENAME FILE

INPUT `DS:DX` contains address of an `ASCIIZ` file name to be renamed
`ES:DI` contains address of an `ASCIIZ` file name which is the new name for the file.

OUTPUT If Carry flag is set then `AX` contains an error code.

57h : GET/SET FILE DATE/TIME

INPUT BX contains open file handle.
AL is set to sub-function code:

= 0	get file date/time
= 1	set file date/time

CX contains new file time (if AL = 1).
DX contains new file date (if AL = 1).

OUTPUT If Carry flag is set then AX contains an error code, otherwise, if AL was 0 on entry:

CX	file time
DX	file date

59h : GET EXTENDED ERROR

INPUT BX must be 0.

OUTPUT
- AX extended error code.
- BH error class.
- BL suggested action.
- CH locus.

NOTE: CL, DX, SI, DI, ES, and DS are destroyed by this function!

5Ah : CREATE UNIQUE FILE

INPUT DS:DX contains the address of an ASCIIZ drive/path string, ending with '\'.
CX contains file attributes.

OUTPUT If Carry flag is set then AX contains an error code, otherwise DS:DX contains the address of a string containing the unique file name and AX contains the open file handle for that file.

5Bh : CREATE NEW FILE

INPUT DS:DX contains the address of an ASCIIZ file name string and CX contains file attributes.

OUTPUT If Carry flag is set then AX contains an error code, otherwise, AX contains the file handle of the newly created, and opened, file.

5Ch : LOCK/UNLOCK FILE BYTES

INPUT AL contains sub-function code:
= 0 lock bytes
= 1 unlock bytes
BX contains file handle.
CX:DX contains the position in the file that the region begins
SI:DI contains the size of region in bytes.

OUTPUT If Carry flag is set then AX contains an error code.

5E00h : GET MACHINE NAME

INPUT DS:DX is the address of a 16-byte buffer.

OUTPUT If Carry flag is set then AX contains an error code. Otherwise if CH is 0, then the machine name/number is undefined. Otherwise, the buffer contains machine name and CL contains the NETBIOS machine number.

5E02h : SET PRINTER SETUP STRING

INPUT `BX` contains the redirection list index.
`CX` contains length of the string (max. 64 bytes).
`DS:SI` contains address of setup string.

OUTPUT If Carry flag is set then `AX` contains an error code.

5E03h : GET PRINTER SETUP STRING

INPUT `BX` contains the redirection list index.
`ES:DI` contains address of a 64-byte buffer.

OUTPUT If Carry flag is set then `AX` contains an error code. Otherwise, `CX` contains the length of the setup string, and the buffer pointed to by `ES:DI` contains the set-up string.

5F02h : GET REDIRECTION LIST ENTRY

INPUT `BX` contains redirection index (0-based).
`DS:SI` contains address of 128-byte buffer (for local name).
`ES:DI` contains address of 128-byte buffer (for network name).

OUTPUT If Carry flag is set then `AX` contains an error code.
Otherwise:

`BH` device status in low-order bit.
= 0 device valid
= 1 device invalid

`BL` device type.

`CX` stored parameter value.

`DS:SI` buffer set to `ASCIIZ` local name.

`ES:DI` buffer set to `ASCIIZ` network name.

NOTE: The contents of `DX` and `BP` are destroyed by this function.

5F03h : REDIRECT DEVICE

INPUT BL contains the device type:
03 printer device
04 file device
CX must be 0.
DS:SI points to an ASCIIZ local name.
ES:DI points to an ASCIIZ network name.

OUTPUT If Carry flag is set then `AX` contains an error code.

5F04h : CANCEL REDIRECTION

INPUT DS:SI points to ASCIIZ local name.
OUTPUT If Carry flag is set then AX contains an error code.

60h : EXPAND PATH

INPUT DS:SI points to an ASCIIZ path string to be expanded.
OUTPUT ES:DI contains the address of the expanded path or filename.

62h : GET PROGRAM SEGMENT PREFIX

INPUT None.
OUTPUT BX contains the segment address of the caller's PSP (i.e. BX:0000 is the address of PSP).

65h : GET EXTENDED COUNTRY

INPUT AL contains the ID for the information required.
BX contains the code page number or FFFFh if the global page is required.
DX contains the country code or FFFFh for the current country.
ES:DI points to a buffer to contain the requested information and CX contains the size of this buffer.

OUTPUT If Carry flag is set then AX contains an error code.
Otherwise CX contains the length of the returned information which is placed in the buffer pointed to by ES:DI.

66h : GET/SET GLOBAL CODE PAGE

INPUT AL specifies the sub-function to be performed:
= 1 get page
= 2 set page
If AL = 2, then BX must be set to the code page you wish to set.

OUTPUT If Carry flag is set then AX contains an error code.
Otherwise, if Al was set to 1 on entry:
BX active code page
DX system code page

67h : SET HANDLE COUNT

INPUT BX contains number of open handles to allow.
OUTPUT If Carry flag is set then AX contains an error code.

68h : FLUSH FILE

INPUT	`BX` contains open file handle.
OUTPUT	If Carry flag is set then `AX` contains an error code.

APPENDIX D
8086/8087 INSTRUCTION SETS

This appendix describes the architecture and instruction set of the 8086 CPU and the 8087 floating-point co-processor. Individual technicalities of these sets are not discussed here; these can be found in, for example, *Intel's 80286 and 80287 Programmer's Reference Manual*.

Architecture

The 8086 CPU (and the 80286 and 80386) contains fourteen 16-bit registers. These registers can be logically split into three main groups:

Data group — consisting of four registers, each 16-bits wide. These are ax, bx, cx and dx. These can each be used as either a single 16-bit register or as two 8-bit registers. When used as 8-bit registers, the two parts are referenced as either the upper half (h) or the lower half (l). The upper 8-bit registers are ah, bh, ch and dh. The lower 8-bit registers are al, bl, cl and dl. The Data Group is used for arithmetic and logical operations.

Pointer and index groups — consisting of sp, bp, ip, si and di. These are all 16-bit registers and generally addresses or address offsets.

Segment group — consisting for four 16-bit segment registers, cs, ds, ss and es. On the 8086 (and the 80286 and 80386 in real mode) these registers are combined with other 16-bit values to obtain (at least) 20-bit wide addresses.

In protected mode on the 80286 and 80386, the segment registers contain selectors which are used by the CPU as indexes into descriptor tables. These tables contain, among other things, the actual address components. These address components are combined with the other 16-bit offset values (possibly 32-bit on the 80386) to obtain a 24-bit address (32-bits on the 80386).

Abbreviation	Meaning
Segment registers	
cs	Code Segment register
ss	Stack Segment register
ds	Data Segment register
es	Extra Segment register
Data registers	
ax	The AX data register (the “accumulator”)
bx	The BX data register
cx	The CX data register
dx	The DX data register
Pointer and index registers	
sp	Stack Pointer
bp	Base Pointer
si	Source Index register
di	Destination Index register
ip	Instruction Pointer
Flag register	
fl	Flag Register

The fourteenth register, fl, is the *Flag Register*. This is treated as 16 1-bit flags, of which only nine are actually used by the 8086 CPU. Table D.2 shows the physical layout of the flag register. The flags themselves are described below. The term “set” means the flag is on (i.e., equal to 1); “cleared” means the flag is off (i.e., set to 0).

Layout of Flags Register (— means the bit is not used)

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-- -- -- -- OF DF IF TF SF ZF -- AF -- PF -- CF
```

Table D.2 Flags Register

OF	Overflow flag: This is set when a signed arithmetic overflow occurs, that is, when a result exceeds the capacity of the destination.
DF	Direction flag: The setting of this flag determines the direction of data transfer. When the flag is set, the string operations which use si and di, decrement these registers in each iteration of the string instruction. When this flag is cleared, si and di are incremented in each iteration of

the string instruction.

IF	Interrupt enable flag: Hardware interrupts can only occur when this flag is set.
TF	Trap flag: When this flag is set, the program single-steps, executing a single instruction. It is for use by debuggers, such as TopSpeed's Visual Interactive Debugger.
SF	Sign flag: This flag is set if the most significant bit of the result of an instruction is also set. Since negative numbers are represented in two's compliment form, this flag is set when the result is negative. If the result is positive, the flag is cleared.
ZF	Zero flag: This flag is set when the result of the instruction is zero. If the result is non-zero, the flag is cleared.
AF	Auxiliary flag: This flag is set by certain instructions when an internal carry has occurred between the lower and upper halves of a byte. If this does not occur, then the flag is cleared. It is not usually useful.
PF	Parity flag: This flag is set when the total number of bits in the result is an even number (even parity). If the number of bits is odd (odd parity) this flag is cleared.
CF	Carry flag: This flag is set if the instruction required an arithmetic carry or borrow in the most significant bit. If this did not occur, the flag is cleared. If unsigned arithmetic is being used, the setting of the carry flag indicates whether or not unsigned overflow occurred.

OF, SF, ZF, AF, PF and CF are set by the arithmetic and logical operations:

- If OF is set it indicates a signed overflow.
- ZF is set if the result is zero overflow otherwise it is cleared.
- If CF is set it indicates an unsigned overflow.

Note: inc and dec do not alter CF.

Data transfers such as mov, push and pop do not alter the flags. For further information about the setting of particular flags by particular instructions, please refer to the Intel documentation. However, this information is not normally required.

Memory Addressing

The segment registers (cs, ds, ss and es) act as base pointers for memory addresses. Every memory access made by a program is relative to one of these registers. The byte (or word) to be accessed has an address which is calculated by the following formula:

```
address = (16 * segment_register_value) + 16_bit_offset
```

This gives a 20-bit physical address on the 8086. This allows up to 1 megabyte of addressable memory on the 8086 processor. Jump addresses are usually relative to the current cs (CODE segment) register value; data is usually relative to the current ds (DATA segment) register value.

If you require a different segment register to address a particular memory item, the segment override syntax should be used:

```
<segment- resister> : [ <offset> ]
```

When any byte is addressed, be it code or data, a segment register is used by the processor. The instruction normally dictates which register is to be used, however, you can define a specific register using the procedure described above.

8086 Instructions

Instructions are fetched from memory at the address currently pointed to by cs:[ip].

After an instruction has been fetched (but before it is executed), the ip register is updated to point to the next instruction. Except for this automatic updating of the next instruction pointer, the only other instructions which affect the cs and ip registers are:

```
jmp       (* and the other jump instructions *)

call
ret

int
iret
```

An 8086 instruction has the following general format:

```
opcode specifier operand1, operand2
```

The *specifier*, *operand1* and *operand2* elements are optional. Their presence or absence depends upon the format for a particular instruction.

The specifier is used to dictate the size of the operands to which the instruction is to be applied. It is used when more than one data size is possible for an instruction and the size is not implied by a register operand. For example, using al implies a byte operand; using ax could apply to both word and byte operands.

The specifiers are:

`byte`	1 byte
`word`	2 bytes

`dword`	4 bytes
`qword`	8 bytes (floating-point only)
`tbyte`	10 bytes (floating-point only)
`near`	2 bytes (for jump, call and return instructions)
`far`	**4 bytes (for jump, call and return instructions)**

8086 Operands

The operand to an 8086 instruction may be one of the following:

- A general register (ax, bx, cx, dx, sp, bp, si, di, al, bl, cl, dl, ah, bh, ch or dh).
- A segment register (cs, ss, ds or es).
- A constant (either a number or a label).
- A memory operand.

These operand types are known, respectively, as R, S, C and M. Table D.3 shows the possible combinations of these types which can occur in 8086 instructions.

Note: the order in which the operand types are given is significant. The combination MC is allowed but the combination CM is not.

Abbreviation	**Description**
Zero Operands	
-	No operand required
One Operand	
R	General register
S	Segment register
C	Constant
M	Memory
Two Operands	
RR	Two general registers
RS	General register;Segment register
RC	General register;Constant
RM	General register;Memory
SR	Segment register;General register
SM	Segment register;Memory
MS	Memory; Segment register
MR	Memory; General register
MC	Memory; Constant
CR	Constant; General register

Table D.3 Allowed 8086 operand combinations

A memory operand is specified by an optional segment override followed by a list of 16-bit offsets, each enclosed in square brackets. A *segment override* is the name of a segment register followed by a colon. For example:

```
es : [Var1] [10]
```

When no segment override is given, ds is used as the segment, unless one of the offsets is the bp register. In this case, the ss register is used as the default segment.Offsets may be labels, numbers or the index registers bx, bp, si or di. There are eight allowed combinations of these index registers:

```
[bx]
[bp]
[si]
[di]
[bx] [si]
[bx] [di]
[bp] [si]
[bp] [di]
```

All other combinations are illegal. However, index registers can be used with labels and numbers to produce such expressions as:

```
mov ax, [_Array] [bp] [si] [10]
```

Instruction Opcode Descriptions

Table D.4, below, summarizes the 8086 instruction set. The instructions are listed in alphabetical order with the following information:

- The opcode mnemonic.
- The possible specifiers which can be used with the instruction.
- The possible operand combinations (see Table D.3) allowed for this instruction.
- A description of the instruction in C-like terms.

The following notation is used in the table to explain actions not easily described:

push(x)	means sp := sp + 2; ss:[sp] = x
pop(x)	means x = ss:[sp]; sp = sp - 2
next(x,i)	means IF (DF) THEN x := x - 1 ELSE x := x + 1

rol(x,y) means rotate x leftwards by y bits

ror(x,y)	means rotate x rightwards by y bits
sar(x,y)	means shift x right y bits, preserving the sign of x
? =	means discard the result

Instructions which discard their results are generally executed for their effect on the flag register. In addition the following points should be noted:

- ‘op1’ and ‘op2’ stand for the first and second operands, respectively.
- Where the specifiers are given as near and far, the cs register is only involved in the far case.
- The notation ‘a:b...’ should be interpreted to mean one of the following:
- A segment override if ‘a’ is a segment register.
- A 32-bit value if ‘a’ is dx.
- A sequence of items in memory with ‘a’ as the high address and with the addresses decreasing as you move along the list.

Comparisons and Jumps

After the comparison instruction:

```
cmp op1, op2
```

the following interpretation of the jump instruction are valid:

```
je    label     (* if (op1 == op2) goto label *)
jne   label     (* if (op1 != op2) goto label *)
```

If unsigned arithmetic is being carried out, then the following can be used:

```
jb    label     (* if (op1 < op2) goto label *)
ja    label     (* if (op1 > op2) goto label *)
jbe   label     (* if (op1 <= op2) goto label *)
jae   label     (* if (op1 >= op2) goto label *)
```

On the other hand, if signed arithmetic is being used:

```
jl    label     (* if (op1 < op2) goto label *)
jg    label     (* if (op1 > op2) goto label *)
jle   label     (* if (op1 <= op2) goto label *)
jge   label     (* if (op1 >= op2) goto label *)
```

Floating-point (8087) Instructions

The 8086 is rather limited when it comes to mathematical operations on anything other than small integers. The 8087 floating-point co-processor extends the abilities of the 8086 to include operations on floating-point numbers up to an accuracy of about 18 decimal digits. In addition, the TopSpeed libraries include emulation of the 8087’s floating point instructions, so you can write assembly language programs including 8087 instructions even if you do not have such a piece of hardware on your computer.

The 8087 has a stack-based architecture with eight 10-byte registers. These are accessed relative to an internal pointer, st, which is updated by some operations. The 8087 registers are referred to by the mnemonics st(0), st(1)

.... st(6), st(7). The first two, st(0) and st(1), have a special status, as can be seen from Table D.7.

In addition to these registers, the 8087 has another 7 registers shown in Table D.5.

Abbreviation	Description
CW	Control Word
SW	Status Word
TW	Tag Word
IPL	Instruction Pointer Low (for the 8087)
IPH	Instruction Pointer High (for the 8087)
DPL	Data Pointer Low
DPH	Data Pointer High

Table D.5 8087 Registers

Generally, these registers are of little interest, and are not discussed here. The only exception is SW, the Status Word, which must be transferred to the 8086's flag register (fl) in order to determine the results of the 8087's comparison instructions.

Table D.6 (overleaf) shows the possible operands for the 8087 instructions.

Abbreviation	Description
Zero Operands	
-	No operand required
One Operand	
T	Stack top (st(0))
R	Any floating-point register (st(0) ... st(7))
M	Memory
Two Operands	
TM	st(0); Memory
MT	Memory; st(0)
TR	st(0); Any floating-point register
RT	Any floating-point register; st(0)
T2	st(1); st(0)
2T	st(0); st(1)

Table D.6 8087 Operand Combinations

Table D.7 lists the floating-point instructions in much the same way as Table B.4 does for the normal 8086 instructions. However, the following points should be noted:

- ‘push’ means —st.
- ‘pop’ means ++st.
- The operand combinations allowed are those listed in Table D.6.
- The specifiers can now include qword, dword and tbyte.
- The operator ‘**’ is used to mean ‘raise to the power of’.

Opcode	Specifiers	Operands	Description
f2xml		T	op1 = (2 ** op1) - 1;
fabs		T	op1 = abs(op1);
fadd	dword qword	TR RT TM	op1 = op1 + op2
faddp		RT	op1 = op1 + op2; pop;
fbld		TM	push; op1 = op2; /* op2 is in packed decimal */
fbstp		TM	op1 = op2; pop; /* op2 is in packed decimal */
fchs		T	op1 = -op1;
fclex			Not described here
fcom	dword qword	TR TM	? = op1 - op2;
fcomp	dword qword	TR TM	? = op1 - op2; pop;
fcompp	dword qword	2T	? = op1 - op2; st += 2;
fdecstp		-	push;
fdiv	dword qword	R RT TM	op1 = op1 / op2;
fdivp		RT	op1 = op1 / op2; pop;
fdivr	dword qword	TR RT TM	op1 = op2 / op1;
fdivrp		RT	op1 = op2 / op1; pop;
ffree			Not described here
fiadd	word dword	TM	op1 = op1 + op2; /* op2 is an int */
ficom	word dword	TM	? = op1 - op2; /* op2 is an int */
ficomp	word dword	TM	? = op1 - op2; pop; /* op2 is an int */
fidiv	word dword	TM	op1 = op1 / op2; /* op2 is an int */
fidivr	word dword	TM	op1 = op2 / op1; /* op2 in an int */
fild	word dword qword	TM	push; op1 = op2; /* op2 is an int */
fimul	word dword	TM	op1 = op1 * op2; /* op2 is an int */
fincstp		-	++st;
finit		-	Initialize the 8087;
fist	word dword	MT	op1 = op2; / op1 is an int */

Table D.7 8087 Opcodes

Opcode	Specifiers	Operands	Description
fistp	word dword qword	MT	op1 = op2; pop; /* op1 is an int */
fisub	word dword	TM	op1 = op1 - op2; /* op2 is an int */
fisubr	word dword	TM	op1 = op2 - op1; /* op2 is an int */
fld	dword qword tbyte	TR TM	temp = op1; push; op2 = temp
fldcw		M	CW = op1;
fldenv		M	DPL:DPL:IPH:IPL:TW:SW: =op1;
fld1		T	push; op1 = 1.0;
fldl2e		T	push; op1 = log2(e);
fldl2t		T	push; op1 = log2(10);
fldlg2		T	push; op1 = log10(2);
fldln2		T	push; op1 = log(2);
fldpi		T	push; op1 = 3.14159;
fldz		T	push; op1 = 0.0;
fmul	dword qword	TR RT TM	op1 = op1 * op2;
fmulp		RT	op1 = op1 * op2; pop;
fpatan		T2	op1 = arctan(op1); pop;
fprem		T	op1 = op1 -(k * st(1)); /* preserving sign */
fptan		T	push; st(1)/st(0)=tan(st(1));
frndint		T	op1 = round(op1);
frstor		M	8087 state = op1(94 bytes);
fsave		M	op1 = 8087 state (94 bytes);
fscale		2T	op1 = op1 * (2 ** op2);
fsqrt		T	op1 = sqrt(op2);
fst	dword qword	RT MT	op1 = op2;
fstcw		M	op1 = CW;
fstenv		M	op1 = DPL:DPL:IPH:IPL:TW:SW:W
fstp	dword qword tbyte	RT MT	op1 = op2; pop;

Table D.7 8087 Opcodes (continued)

Opcode	Specifiers	Operands	Description
stsw		M	op1 = SW;
fsub	dword qword	RT TR TM	op1 = op1 - op2;
fsubp		RT	op1 = op1 - op2;
fsubr	dword qword	RT TR TM	op1 = op2 - op1;
fsubrp		RT	op1 = op2 - op1; pop;
ftst		T	? = st - 0.0;
fwait			wait for completion
fxam		T	Not described here

fxch		TR RT	temp = op1;
			op1 = op2;
			op2 = temp;

fxtract		T	push; st(0) * (2 ** st(1)) = st(1);
fyl2x		T2	st(1) = st(1) * log2(st(0)); pop;
fyl2xp1		T2	st(1) = st(1) * log2(st(0) + 1.0); pop;

Table D.7 8087 Opcodes

Using Floating-point Instructions

To illustrate the use of floating-point instructions, and to amplify on the use of the foption keyword, here is a simple example which adds two integers:

```
module ftest

segment _TEXT (CODE,28H)
select _TEXT

entry:
  jmp prog

  Number1 : dw 4                (* define 1-word integers *)
  Number2 : dw 3
  Answer  : dw 0

prog:
  finit                        (* Initialize the 8087 *)
  fild  word st(0), [Number1]             (* load int to top of stack *)
  fiadd word st(0), [Number2]
              (* integer add *)
  fistp word [Answer], st(0)
              (* integer store *)
  fwait                        (* wait for 8087 to finish *)
  int 20H

end
```

Notice the syntax and specifiers which are used in this example. If you examine the resulting code with DEBUG or the TopSpeed Disassembler, the object code generated will take the following form:

```
WAIT
  FINIT
WAIT
  FILD    WORD PTR [0002]
WAIT
  FIADD   WORD PTR [0004]
WAIT
  FISTP   WORD PTR [0006]
```

Note: The WAIT instructions are inserted by the assembler in order to synchronize the 8087 and the 8086. This is required because the 8086 and the 8087 are executing instructions in parallel. If you use foption 2, the wait states are suppressed.

Floating-point Comparisons and Jumps

In order to load the 8086's flag register with the results of a floating-point comparison on the 8087, the following sequence of operations must be carried out:

```
fcom  op1, op2  (* Comparison *)
fstsw [temp]      (* Store 8087 status word *)
fwait
mov ax, [temp]  (* Status word into AX *)
sahf            (* Set comparison results flags *)
```

The following jump instructions are now valid:

```
je    label     (* if (op1 == op2) goto label *)
jne label     (* if (op1 != op2) goto label *)
jb    label     (* if (op1 < op2) goto label *)
ja    label     (* if (op1 > op2) goto label *)
jbe label     (* if (op1 <= op2) goto label *)
jae label     (* if (op1 >= op2) goto label *)
```

Note: The jumps associated with floating-point arithmetic are those normally used with unsigned arithmetic. This can cause problems for those unexperienced in 8087 programming, as the expectation is that the signed jump instructions should be used.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U