

TechKit

TopSpeed Modula-2



Eric Striel

TopSpeedTM Modula-2

For IBM[®] Personal Computers and Compatibles

TechKit (Revised Edition)

Jensen & Partners International

LICENSE STATEMENT

Jensen & Partners International hereby grants you a non-exclusive license to use the TopSpeed Modula-2 TechKit. You may use the software on any computer, **provided that the software cannot possibly be used on more than one computer at the same time.** You may make copies of the software for the sole purpose of loading it into the computer on which you wish to use it, or for keeping a maximum of two backup copies.

You are furthermore granted the right to sell programs developed by you using the program modules provided with this package. You are not allowed to resell or give away any part of the source code provided with this product.

WARRANTY

Jensen & Partners International warrants the enclosed diskette to be an accurate copy of the master disk, and will replace any defective copy free of charge for a period of 60 days after purchase.

Jensen & Partners International hereby explicitly disclaims all other warranties whether express or implied, including without limitation, the implied warranties of merchantability and fitness for any particular purpose. Jensen & Partners International shall have no liability for consequential, incidental, exemplary or special damages, including lost profits.

Jensen & Partners International reserves the right to change the product at any time, without prior notice.

TopSpeed™ is a trademark of Jensen & Partners International.

IBM® is a registered trademark of the International Business Machines Corporation.

Microsoft® is a registered trademark of Microsoft Corporation.

GEM™ is a trademark of Digital Research Inc.

SideKick® is a registered trademark of Borland International, Inc.

BIX® is a registered trademark of McGraw-Hill, Inc.

Prokey™ is a trademark of RoseSoft, Inc.

Copyright © 1988, 1989 by Jensen & Partners International. All rights reserved.

Printed in the United States of America.

Contents

1	Introduction	1
2	Controlling Code and Modules.....	3
	Context.....	4
	Default Storage Layout	4
	Controlling Code and Data Segmentation	9
	Controlling Run-Time Library Calls.....	10
	Near and Far Calls	12
	Controlling the Order of Classes	15
	Creating Code with Absolute Addresses	15
	Putting Things in (P)ROM.....	16
	8087 Support	17
3	Using Overlays in Modula-2 Programs.....	19
	Introduction to Overlays	19
	Using the OVL Program	20
	Handling Small Segments and Other Worries.....	21
	Linking Large Programs	21
	Use the Batch Linker	22
	Structure Global Variables	22
	Reduce the Number of Segments	22
	Decrease the Number of Procedures	22
	Suppress Libraries When Possible	22
	Using the OVLEXE Program	23
	Example	23
	Restrictions on the Use of Overlays	25
	OVLEXE Warnings.....	25
	Implicit Assignment Rules	26
	Operation of OVLEXE	26
	Options	27
	The OvlMan Module	28
	OVLMAP and OVLXREF	28
	Debugging Overlays	29
	More Technical Data	29

4	8088/8087 Instruction Sets	33
	Registers	33
	Memory Addressing	35
	Instructions	36
	Operands	36
	Opcode Descriptions	38
	Floating Point (8087) Instructions	42
5	The TopSpeed Assembly Language	47
	How to Call the Assembler	48
	Program Style	48
	Segments	49
	Example: Setting Up a Code Segment	50
	Example: Setting Up a Data Segment	50
	Tokens	50
	Generic Tokens	50
	Keywords	51
	Operator Precedence	51
	Syntax	51
	Instructions	54
	Symbol Table	55
	Operand Sizes	56
	Jumps and Calls	56
	Strings	56
	References	56
	Non-8086 Instructions	56
	Variables	57
	Miscellaneous Points	57
	Naming Conventions	58
	TopSpeed Assembler Error Messages	58
6	Interfacing Assembler to Other Languages	65
	Calling Conventions	66
	A Short Example	66
	Assembling the File	67
	Initialization Code for Assembler Modules	68
	A More Complex Example	68
	Using Naughty	71
	Storage Layout for the Naut Program	71
7	Creating TSR Programs	73
	Activating a TSR Program	74
	Installing a TSR Program	75
	A Short TSR Program	76

Cautions about TSR Programs	77
Deactivating a TSR Program	78
Terminating TSR Programs	78
The TSR Module	79
The TSRCALC Program	79
8 Using the RS-232 Port	81
The rs.Def File	82
Initializing the Port	82
Keeping Track of Buffers	84
Sending and Receiving	85
Sending and Checking for Breaks	86
An Example Program	87
The RDemo Program	87
9 Miscellaneous Modules	89
BiosIO Module	89
A Critical Error Handler	91
A Run-Time Error Handler	92
Using Lotus/Intel/Microsoft Expanded Memory	94
10 Programs Included With the TechKit	99
Analyze	99
ObjCompr	100
SetHeapSize	101
Conclusion	101
Index	103

Eric Striel

TopSpeedTM Modula-2

For IBM[®] Personal Computers and Compatibles

TechKit (Revised Edition)

Jensen & Partners International

Contents

1	Introduction	1
2	Controlling Code and Modules.....	3
	Context.....	4
	Default Storage Layout	4
	Controlling Code and Data Segmentation	9
	Controlling Run-Time Library Calls.....	10
	Near and Far Calls	12
	Controlling the Order of Classes	15
	Creating Code with Absolute Addresses	15
	Putting Things in (P)ROM.....	16
	8087 Support	17
3	Using Overlays in Modula-2 Programs.....	19
	Introduction to Overlays	19
	Using the OVL Program	20
	Handling Small Segments and Other Worries.....	21
	Linking Large Programs	21
	Use the Batch Linker	22
	Structure Global Variables	22
	Reduce the Number of Segments	22
	Decrease the Number of Procedures	22
	Suppress Libraries When Possible	22
	Using the OVLEXE Program	23
	Example	23
	Restrictions on the Use of Overlays	25
	OVLEXE Warnings.....	25
	Implicit Assignment Rules	26
	Operation of OVLEXE	26
	Options	27
	The OvlMan Module	28
	OVLMAP and OVLXREF	28
	Debugging Overlays	29
	More Technical Data	29

Cautions about TSR Programs	77
Deactivating a TSR Program	78
Terminating TSR Programs	78
The TSR Module	79
The TSRCALC Program	79
8 Using the RS-232 Port	81
The rs.Def File	82
Initializing the Port	82
Keeping Track of Buffers	84
Sending and Receiving	85
Sending and Checking for Breaks	86
An Example Program	87
The RDemo Program	87
9 Miscellaneous Modules	89
BiosIO Module	89
A Critical Error Handler	91
A Run-Time Error Handler	92
Using Lotus/Intel/Microsoft Expanded Memory	94
10 Programs Included With the TechKit	99
Analyze	99
ObjCompr	100
SetHeapSize	101
Conclusion	101
Index	103

Chapter 1

Introduction

Welcome to the TopSpeed Modula-2 TechKit — a collection of utilities and resources to enable you to make even more effective use of your TopSpeed Modula-2 system. The resources in the TechKit will allow you to do more “advanced” things, such as

- writing terminate and stay resident (TSR) programs
- modifying the run-time library or other modules written in assembler
- using the RS-232 port for serial communications
- and even using the benefits provided by expanded memory that conforms to the Lotus/Intel/Microsoft (LIM) standard

The TechKit includes **M2ASM**, an assembler. It also includes modules that enable you to use overlays from within your programs, a program for putting code in absolute memory locations (for example, if you need to run a program that is stored in a PROM device), programs to analyze module dependencies, and so forth. In addition, the TechKit contains a variety of source code — including the run-time library and **AsmLib** assembler modules.

There are a lot of very useful tools and examples in the TechKit, and these resources will enable you to do a considerable number of impressive things. However, to take full advantage of these tools, you should be comfortable working with your computer at fairly low levels.

For example, you should be reasonably competent at working with DOS. You should understand the operating system's methods for memory layout and management, and the use of interrupts before you try such things as creating TSR programs or using expanded memory. The discussions in this manual also assume you are familiar with fundamental aspects of program and memory layout, and with the use of registers and interrupts.

This manual covers the following topics:

- Controlling code layout and segmentation
- Creating code with absolute addresses
- Using overlays in your programs
- 8087 support
- The TopSpeed Assembler (M2ASM) and assembly language
- Using assembler modules and routines in non-assembler programs
- Creating terminate-and-stay-resident programs
- Using the RS-232 port
- Miscellaneous modules included in the TechKit
- Miscellaneous programs included in the TechKit

Controlling Code and Modules

Once you've written your Modula-2 program, the TopSpeed compiler components will do everything else needed to convert your source file into a running program. The **Make** facility will ensure that all the various files used to build your program are the most recent versions of those files. The compiler will then translate your program into an efficient, fast and compact form. This compiled version will be used by the linker in the next step. The linker will run through the code produced by the compiler and automatically find and read in all the procedures used from the relevant modules, producing an **.EXE** file that contains only the procedures required for your program.

If you need to, you can also take control of these activities — to build just the kind of program or environment you want. This can be very useful if you need to write specialized programs (such as embedded systems, programs that draw on utilities developed in other languages, and so forth), or if you need to keep the program size down.

The TopSpeed Modula-2 environment makes it easy to exert such control. To help you take better advantage of your options, a brief summary of how an **.OBJ** file is laid out under various conditions is included. To illustrate this, let's look at the storage layout for a few examples in which various options are activated for a program. See Chapter 7 of the *TopSpeed Modula-2 User's Manual*, for general background and additional details about the compiler.

Context

The linker automatically determines the files that will be needed when creating a particular program file. The information that allows it to do this is derived from a topological sort of the modules. Such a sort arranges the modules with respect to their **IMPORT** dependencies.

Modules that have no dependencies come first in such a sort, followed by their client modules. When the linker arranges the initialization code for the modules, the code is arranged in a sequence based on the topological sort. This method of working ensures that the initialization code for a certain module is executed before that particular module is called by another module. All initialization code is put into the same segment, named **INITCODE**.

Default Storage Layout

There are three main categories into which all code and data associated with a program are aligned and ordered.

- A **SEGMENT** contains up to 64Kb of (discrete) code or data, and is referred to by a name. Every code or data unit belongs within a segment. Each module must contain all its global data within a segment — therefore there is a 64Kb limit on the global data for any module. (The code for a module must be contained within a single group.) The linker ensures that all the items that are to be in the same segment are within the same 64Kb of RAM.
- A **GROUP** contains up to 64Kb worth of related segments, and also carries a name with which it can be identified. A group can contain more than one segment, but a segment does not necessarily have to belong to a group. Putting a number of segments within a group tells the linker that they must all be referenced using the same segment register.
- A **CLASS** contains a number of segments, and takes priority in naming and ordering conventions. That is, information will be sorted primarily by class name, then by segment name (so segments within different classes may have the same name). Classes are ordered (by the linker) according to their sequence. This means that for good optimization and efficiency it is up to the compiler to write the classes in correct sequence.

The following listing will help illustrate the storage layout of a TopSpeed Modula-2 program:


```

DEFINITION MODULE x;
VAR I : INTEGER;
PROCEDURE p;
END x.

-----

IMPLEMENTATION MODULE x;
CONST S='THIS IS A STRING CONSTANT';
PROCEDURE p;
BEGIN
  I := INTEGER(S[6]);
END p;
BEGIN
  I := 0;  o 7  3
END x.

-----

MODULE y;
IMPORT x;
PROCEDURE q;
BEGIN
  x.p;
END q;

BEGIN
  IF x.I = 0 THEN x.p END;
END y.

```

The linker map file shows the following storage layout for this example:

Start	Stop	Length	Name	Class	Group
00000H	0006AH	0006AH	INITCODE	CODE	G_CODE
0006AH	00539H	004CFH	C_CODE	CODE	G_CODE
00539H	00540H	00007H	S_CODE	CODE	G_CODE
00540H	00552H	00012H	C_Y	FCODE	G_Y
00552H	00554H	00002H	S_Y	FCODE	G_Y
00554H	00554H	00000H	K_Y	FCODE	G_Y
00554H	00577H	00023H	C_X	FCODE	G_X
00577H	00579H	00002H	S_X	FCODE	G_X
00579H	00593H	0001AH	K_X	FCODE	G_X
00593H	00593H	00000H	DATA	DATA	DGROUP
005A0H	005A0H	00000H	DATASTART	DUMMY	DGROUP
005A0H	005A0H	00000H	BSS	BSS	DGROUP
005A0H	005A0H	00000H	D_Y	M_DATA	(none)
005A0H	005A2H	00002H	D_X	M_DATA	(none)
005B0H	045B0H	04000H	STACK	STACK	(none)
045B0H	045B0H	00000H	HEAP	HEAP	(none)

During compilation, four segments are generated for each module, along with a group which contains three of them. The four segments contain: code for any procedures; structured constants; special constants; global variables. These are called:

- C_<module name>** This segment contains the code for the module.
- K_<module name>** This segment contains structured and **REAL** constants.
- S_<module name>** This contains any special segment constants.

D_<module name> This last segment contains the global variables of the module. (This is usually located at the end of a map, just before the final segments of a program: the STACK and HEAP.)

The **C_<module name>**, **S_<module name>**, and **K_<module name>** segments are contained within a group, **G_<module name>**.

By default, the linker groups the program segments along the following lines: initialization code, segments for the run-time system, and segments for individual modules (including data segments). The details of the segment arrangement within these three categories will depend on whether you use any compiler or linker directives when processing the file.

Running through the map-file above: The first three segments contain the initialization code and the run-time library code used in the program. The **S_CODE** segment contains constants used in the **C_CODE** segment. These segments all belong to the class **CODE**, which ensures that these segments are located next to each other when the linker lays out and allocates the storage for the program.

These three segments also belong to the same group, **G_CODE**. When a number of segments are within a group, this instructs the linker to use the same segment register when referencing each segment. This means that when code in segment **C_X** (from the preceding example) is executing, the code can refer to constants in **S_X** and **K_X** using the **CS** (code segment) register.

Similarly, the six segments containing code and constants for modules **x** and **y** all belong to the **FCODE** class. The relevant segments for each of the separate modules are laid out in sequence, but with the data segment (**D_**) following after all the **C_**, **K_**, **S_** segments for each module.

The layout is similar for each module and although a program will have three such segments for each independent module, some of these segments may, in a particular program, be empty.

After the three segments previously described, that is the **C_**, **K_**, **S_** segments, the program data segments come next. The first of these, **_DATA**, is included to make it possible to include code from C compilers. Unless you are using such code, this segment will be empty, as it is for the preceding example. Another data segment, **_BSS**, is also provided, and would be included if C code were be used.

The actual data segments used by programs created with TopSpeed Modula-2 begin with **_DATASTART**. This segment serves to mark the starting location of the uninitialized data segments explicitly. This makes it easy to clear memory quickly and correctly when you use the **\$Z+** compiler directive to initialize all global variables.

The program also has global data segments for each of the modules — **D_Y** and **D_X**. Again, there will be such a segment for each independent module. Names for global data segments begin with **D_**. By default, this prefix is followed by the module name.

The final two segments contain the main program's stack and the heap available for dynamic storage. The program will use the stack to do most of its work, and will get space from the heap as requested in the program — for example, through a call to **ALLOCATE**. (The heap segment marks the start of the heap. Normally, a program will use all of memory above the heap segment as the program's heap.)

When the TopSpeed Modula-2 compiler produces an **.OBJ** file, the variables and procedures within this file refer to each other by public external names. These names follow certain predefined naming and calling conventions. These should be noted and followed, especially when using the assembler.

The naming conventions are:

- Initialization code for each module, as within the **INITCODE** segment, is referenced as the module name followed by an **@**-sign. A module called **Search** would have initialization code called **Search@**.
- All variable names are prefixed with the module name and an **@**-sign. This means a variable **var1** in module **Search** would be referred to in an **.OBJ** file as **Search@var1**.
- References to procedures are dependent on whether they require **NEAR** or **FAR** calls. **FAR** calls are referred to by the module name, a **\$**-sign, then the procedure name. So a procedure **Intr** in the **AsmLib** module would, as a **FAR** call be referred to as **AsmLib\$Intr**. **NEAR** calls are similar but use an **@**-sign. That is, procedure **FoundIt** in module **Search** would be **Search@FoundIt**.

NOTE: Using the **\$G** compiler switch will disable the **<module>\$** or **<module>@** prefixing used with procedures.

The following table summarizes the storage layout for files created by the TopSpeed Modula-2 linker.

DATA	This segment contains the data belonging to the run-time system.
D_Y	This segment contains the variables defined in module Y; in this case none.
D_X	This contains the variables (in this case, 7) defined in module X. Data segments for other modules would follow.
STACK	This segment contains the stack for the main task.
HEAP	This segment marks the start of the rest of memory available for dynamic storage.

Segment	Description
---------	-------------

LOW ADDRESS

INITCODE

This segment contains code for the initialization of the various modules. This can be the actual code (in case of a small code model), or a call to a procedure in a different segment (large code model).

C_CODE

This segment contains the code, belonging to the runtime library and module **SYSTEM**, which can use the small code model. Small and large code model may be mixed. If they are mixed the result is, of course, a large code model.

S_CODE

This segment contains segment constants which are used in the segment **C_CODE**. This segment is included to provide information to reach data stored within the **D_DATA** segment, because segment registers in the 80x86 family cannot be loaded directly with a constant value.

(K_CODE)

If this segment is present, it contains any structured and **REAL** constants defined in a module with code in **C_CODE**. Constants are kept in this segment, separate from variables, because it is more efficient when initializing variables.

C_Y

The code for procedures in module **y** goes here.

S_Y

The segment constants used to refer to the variable **I** (which is in segment **D_Y**) are located here.

K_Y

Since module **y** does not contain any structured constants or **REAL** constants, this segment is empty.

C_X

Similar to **C_Y**, but for module **x**.

S_X

Similar to **S_Y**, but for module **x**.

K_X

Similar to **K_Y**, but for module **x**. Not empty in the example.

_DATA

This segment and the segment **BSS** below are where data relating to any imported C compiler code would be located.

_DATASTART

This segment is used by the compiler (when **\$Z+** is specified in the main module) to clear out all global variables in the program.

_BSS

See **_DATA** above.

D_DATA

This segment contains the data belonging to the run-time system.

D_Y

This segment contains the variables defined in module **y**, in this case none.

D_X

This contains the variables (in this case, **I**) defined in module **x**. Data segments for other modules would follow.

STACK

This segment contains the stack for the main task.

HEAP

This segment marks the start of the rest of memory available for dynamic storage.

HIGH ADDRESS

Controlling Code and Data Segmentation

You can use compiler directives to control how code and data are grouped into segments. For example you can group the data from different modules in the same segment.

The following program illustrates how to do this for both code and data. The `$D` directive, followed by a name, specifies that data for the module should be included in the segment with the same name as the argument — in this case, `y`. Thus, the data for modules `x` and `y` will be put into the same data segment, which will be named `D_y`.

Similarly, the `$M` directive instructs the compiler to put all *code* for modules `x` and `y` in the code segment `C_y`. At the same time, the items for the `S_x` and `K_x` segments are included in `S_y` and `K_y`, respectively, and they are grouped into `G_y`.

```
(* $D y, M y*)
DEFINITION MODULE x;
VAR I : INTEGER;
```

```
PROCEDURE p;
END x.
```

```
-----

(* $D y, M y*)
IMPLEMENTATION MODULE x;
CONST S='THIS IS A STRING CONSTANT';
```

```
PROCEDURE p;
BEGIN
  I := INTEGER(S[6]);
END p;
```

```
BEGIN
  I := 0;
END x.
```

```
-----

MODULE y;
IMPORT x;
```

```
PROCEDURE q;
BEGIN
  x.p;
END q;

BEGIN
  IF x.I = 0 THEN x.p END;
END y.
```


For this example the linker map shows the following storage layout:

Start	Stop	Length	Name	Class	Group
00000H	0006AH	0006AH	INITCODE	CODE	G_CODE
0006AH	00539H	004CFH	C_CODE	CODE	G_CODE
00539H	00540H	00007H	S_CODE	CODE	G_CODE
00540H	00575H	00035H	C_Y	FCODE	G_Y
00575H	00579H	00004H	S_Y	FCODE	G_Y
00579H	00593H	0001AH	K_Y	FCODE	G_Y
00593H	00593H	00000H	DATA	DATA	DGROUP
005A0H	005A0H	00000H	DATASTART	DUMMY	DGROUP
005A0H	005A0H	00000H	BSS	BSS	DGROUP
005A0H	005A2H	00002H	D_Y	M DATA	(none)
005B0H	045B0H	04000H	STACK	STACK	(none)
045B0H	045B0H	00000H	HEAP	HEAP	(none)

Notice that because of the compiler directives used, this time there are only three segments in the **FCODE** class, and only one global data segment for the modules (**D_Y**). The code for modules **x** and **y** now share the segment **C_Y** and the data for these modules share the segment **D_Y**. This is useful because it makes it possible to call between modules **x** and **y** with near calls! The resulting code file will therefore be smaller and faster.

NOTE: If you use the **\$M** directive for an implementation module, you must use the same directive in the definition module.

Controlling Run-Time Library Calls

The following listing shows yet another way to control the layout of your program — this time by influencing the organization of the module code in relation to the run-time library code.

```
(* $D y, M CODE*)
DEFINITION MODULE x;
VAR I : INTEGER;

PROCEDURE p;
END x.

-----

(* $D y, M CODE*)
IMPLEMENTATION MODULE x;
CONST S='THIS IS A STRING CONSTANT';

PROCEDURE p;
BEGIN
  I := INTEGER(S[6]);
END p;
```



```

BEGIN
  I := 0;
END x.

-----

(*$M CODE*)
MODULE y;
IMPORT x;

PROCEDURE q;
BEGIN x.p;
END q;

BEGIN
  IF x.I = 0 THEN x.p END;
END y.

```

In this program, data for the two modules is again put in the same segment, `D_Y`. This time, however, the `$M` directive instructs the compiler that the *code* for both modules is to be put in the same segment as the code for the run-time library — `C_CODE`. Using `CODE` as the segment name argument for `$M` accomplishes this. This option now leads to the following storage layout.

Start	Stop	Length	Name	Class	Group
00000H	00079H	00079H	INITCODE	CODE	G_CODE
00079H	00560H	004E7H	C_CODE	CODE	G_CODE
00560H	00570H	00010H	S_CODE	CODE	G_CODE
00570H	0058AH	0001AH	K_CODE	CODE	G_CODE
0058AH	0058AH	00000H	DATA	DATA	DGROUP
00590H	00590H	00000H	DATASTART	DUMMY	DGROUP
00590H	00590H	00000H	BSS	BSS	DGROUP
00590H	00592H	00002H	D_Y	M DATA	(none)
005A0H	045A0H	04000H	STACK	STACK	(none)
045A0H	045A0H	00000H	HEAP	HEAP	(none)

You will notice that this program has relatively few segments. In particular, the segments previously in class `FCODE` have disappeared, since these program components have been moved into the `C_CODE`, `S_CODE` and `K_CODE` segments.

Because the code for these modules now resides in the same segments as the run-time library code, all calls from these modules to the run-time library can now be achieved with short calls, which of course makes the code smaller and faster. However, although `NEAR` code can call `FAR` code, the reverse is not possible. This scenario, with module code in the same segment as run-time library code, is something of a special case of the code segment option.

Near and Far Calls

An address for a NEAR call is a 16-bit offset relative to the current code segment CS. This means that, due to the minimal arithmetic and manipulation required, a NEAR call results in faster executing and more compact code.

A FAR call uses a 32-bit address arrived at by adding a 16-bit offset to a 16-bit segment register, this requires code segment address manipulation, and generally produces code that executes more slowly.

Another pair of examples will show even more explicitly the consequences of specifying how the program is to be segmented. The following programs differ in the way the main program module (**Test**) is handled.

In the first case, the default layout is used. Notice, however, that procedure **q** in module **Test** uses the **\$N** (NEAR) directive, so that procedures are called with NEAR calls. This means any such procedures can only be called from within the same segment (which, in the example, is **C_TEST**).

```
DEFINITION MODULE Imp;
```

```
  PROCEDURE r;  
  BEGIN  
  END Imp.
```

```
-----  
IMPLEMENTATION MODULE Imp;  
VAR i : CARDINAL;
```

```
  PROCEDURE r;  
  BEGIN  
  END r;
```

```
  BEGIN  
    i := 0;  
  END Imp.
```

```
-----  
MODULE Test;  
IMPORT Imp;  
VAR b : BOOLEAN;  
    11,12 : LONGCARD;
```

```
  PROCEDURE p;  
  BEGIN  
    11 := 11*12;  
  END p;
```

```
  (*$N*)  
  PROCEDURE q;  
  BEGIN  
    11 := 11*12;  
  END q;
```



```

BEGIN
  IF b THEN p; q; END;
END Test.

```

This program has the following layout:

Start	Stop	Length	Name	Class	Group
00000H	00011H	00011H	INITCODE	CODE	G_CODE
00011H	0047CH	0046BH	C_CODE	CODE	G_CODE
0047CH	0048CH	00010H	ENTERCODE	CODE	(none)
0048CH	00490H	00004H	S_CODE	CODE	G_CODE
00490H	004FAH	0006AH	C_Test	FCODE	G_Test
004FAH	004FCH	00002H	S_Test	FCODE	G_Test
004FCH	004FCH	00000H	K_Test	FCODE	G_Test
004FCH	00504H	00008H	C_Imp	FCODE	G_Imp
00504H	00506H	00002H	S_Imp	FCODE	G_Imp
00506H	00506H	00000H	K_Imp	FCODE	G_Imp
00510H	00510H	00000H	DataStart	Dummy	DGROUP
00510H	0051AH	0000AH	D_Test	M_DATA	(none)
0051AH	0051CH	00002H	D_Imp	M_DATA	(none)
00520H	04520H	04000H	STACK	STACK	(none)
04520H	04520H	00000H	HEAP	HEAP	(none)

Test and **Imp** each have their own code and data segments. This program produces the following initialization code:

```

initcode :
38D6:0000  9A0C002539      call  imp@
38D6:0005  9A56001F39      call  test@
38D6:000A  9ADB01D638      call  standard$halt

```

In the default layout the initialization code simply contains (far) calls to the modules. Otherwise, the code for the initialization would need special treatment when addressing constants, since the local constant would no longer be in the same group as the code.

\$MThe next program uses the **\$M CODE** directive to put the code for **Test** in the same segment as the run-time library. A call to **Imp.r** has also been added to help make the main point easier to see.

```

DEFINITION MODULE Imp;

```

```

  PROCEDURE r;
END Imp.

```

```

-----
IMPLEMENTATION MODULE Imp;
VAR i : CARDINAL;

```

```

  PROCEDURE r;
  END r;

```

```

  BEGIN
    i := 0;
  END Imp.
  -----

```



```

(*$M CODE*)
MODULE Test;
IMPORT Imp;
VAR b : BOOLEAN;
    l1,l2 : LONGCARD;

PROCEDURE p;
BEGIN
    l1 := l1*l2;
END p;

(*$N*)
PROCEDURE q;
BEGIN
    l1 := l1*l2;
END q;

BEGIN
    IF b THEN p; q; Imp.r; END;
END Test.

```

The layout for this program is as follows:

Start	Stop	Length	Name	Class	Group
00000H	00022H	00022H	INITCODE	CODE	G_CODE
00022H	004DFH	004BDH	C_CODE	CODE	G_CODE
004DFH	004F0H	00011H	S_CODE	CODE	G_CODE
004F0H	004F0H	00000H	K_CODE	CODE	G_CODE
004F0H	00500H	00010H	ENTERCODE	CODE	(none)
00500H	0050FH	0000FH	C_Imp	FCODE	G_Imp
0050FH	00511H	00002H	S_Imp	FCODE	G_Imp
00511H	00511H	00000H	K_Imp	FCODE	G_Imp
00520H	00520H	00000H	DataStart	Dummy	DGROUP
00520H	0052AH	0000AH	D_Test	M_DATA	(none)
0052AH	0052CH	00002H	D_Imp	M_DATA	(none)
00530H	04530H	04000H	STACK	STACK	(none)
04530H	04530H	00000H	HEAP	HEAP	(none)

Notice that the `C_Test`, `S_Test`, and `K_Test` are no longer needed, since the code for `Test` is now in the segments ending in `_CODE`. The initialization code for this program shows the effect of using the `$M` directive, with `CODE` as its argument.

initcode:			
38D6:0000	9A07002639		call Imp@
38D6:0005	2E8E1EDF04	Test@	mov ds,cs:TestD_Test
38D6:000A	803E000000		cmp b,[0],0
38D6:000F	740C		jz 1D
38D6:0011	0E		push cs
38D6:0012	E82D04		call Test\$p
38D6:0015	E85304		call Test@q
38D6:0018	9A00002639		call Imp\$r
38D6:001D	E8CC01		call Standard@HALT

In this case, instead of a far call to `Test@`, the actual code of `Test`'s initialization is included in the `INITCODE` segment.

Controlling the Order of Classes

Certain operating environments (such as Digital Research's GEM environment) require the program entry point to be at the start of the **.EXE** file. You can accomplish this by assembling and linking in the file **ORDER.A**, which is included with the TechKit for your TopSpeed Modula-2 package.

First, use the assembler included with your TechKit to create an object file. The following command will do this:

```
m2asm order.a
```

To use **ORDER** when creating your program file (**PROG**), use the following batch linker command:

```
m2 /l order+prog, prog /m/c
```

Creating Code with Absolute Addresses

Your TechKit includes a program, **M2LOCATE.EXE**, that enables you to specify exactly where the various segments of your program are to be placed. This can be very useful for creating embedded programs, for putting your programs in PROMs or for making a bootable disk for a program that is to run without DOS.

M2LOCATE is used to locate an MSDOS **.EXE** file. That is, the segments are allocated absolute segment values in the 8086 1-Mbyte memory space. The output is the binary image of the program.

The memory layout for the target system is specified in the **.MEM** file, together with the linker classes which are to be put in the memory. For example:

```
ram 100 60000 DATA Dummy BSS M_DATA STACK HEAP
rom F8000 FFFF0 CODE FCODE
rom FFFF0 100000 BOOTCODE
width 2
```

Each line of the **.MEM** file must start with a **rom**, **ram** or **width** command.

For **rom** and **ram** lines, these words are followed by two hexadecimal numbers, specifying the absolute starting and ending addresses of some memory in the target system configuration. The first hex number is the absolute start address, the second address is a 'stop' address, so that a warning can be produced if there is not enough memory. The addresses are followed by a list of class names, separated by spaces. The order of the names is not significant since the ordering is determined by the order during the linking process.

The word **width** is followed by a number which should be 1, 2 or 4. The default is 1. This number specifies the target data bus width, in bytes. For example, **width 2** indicates that the target data bus is 2 bytes wide, and causes two binary files to be produced — one with the even bytes, one with the odd bytes. This is necessary because PROMs are normally only 1 byte wide, so in a system with a 16-bit data bus at least 2 PROMs will be required.

M2LOCATE is invoked with a single parameter — the file name, but *without* including the extension. For example

```
m2locate myprog
```

When invoked, **M2LOCATE** uses three input files:

- MYPROG.EXE** the MS-DOS executable file
- MYPROG.MAP** the map produced by the linker
- MYPROG.MEM** the ASCII file specifying the memory layout

The output files from **M2LOCATE** are:

- MYPROG.BN?** the binary image(s)
- MYPROG.MP2** the **.MAP** file with edited addresses

Certain environments may have unusual code location requirements. One hint to keep in mind when dealing with such cases is the following: If the class names in the map file prove to be too “coarse,” you can always edit the map file. For example, code segments **C_X** and **C_Y** — which are for modules **X** and **Y**, respectively — would both be in the **FCODE** class, as in previous examples. If your code location requirements dictate that these segments be in different locations, you can change the class names associated with the two segments — by editing the **.MAP** file.

The class for segments belonging to the same group must, however, be the same. Thus, **S_X** and **K_X** should end up in the same class as **C_X**.

Putting Things in (P)ROM

If you need to put code into ROM or into PROM, it's important to keep in mind that **AsmLib** and **SYSTEM** contain self-modifying code. Most of the procedures in **AsmLib** with this property are system dependent. These procedures are *not* necessary for creating a normal, sequential program.

8087 Support

The TopSpeed Modula-2 compiler generates 8087 instructions for floating point operations. At run-time, these instructions either will be executed by an actual 80x87 chip or they will be emulated by software.

The emulator is included if any part of the program uses floating point instructions. At run-time the emulator will check if an 80x87 chip is present. If so, the emulator will patch the emulator calls back to 8087 instructions. The same **.EXE** file can thus be run on any machine at maximum possible speed.

The TechKit also contains a dummy emulator that can help save space. If an 80x87 chip is known to be present, a dummy emulator can be supplied at link-time to save space. This is done by replacing the **SYSTEM.OBJ** file with an **.OBJ** file that defines the necessary public external names to zero. (The **SYSTEM.7BJ** file does this.)

The TopSpeed Modula-2 TechKit includes one overlay management program. This chapter contains a brief description of the systems available for creating and using overlays. The **OVL** and **OVLX86** programs are discussed in separate sections.

Introduction to Overlays

The purpose of a program overlay system is to reduce the amount of RAM that an application program requires in order to run. This is accomplished by splitting the application program into a main part and one or more overlays. The application program then loads the overlays only when it needs to execute some code in them.

This really works well only if the following two conditions hold:

- each overlay has a big enough task so that the time to load the overlay is insignificant in comparison with the amount of work that the overlay has to do (otherwise, it is very slow)
- at least two of the overlays are big enough so that having only one of them at a time in memory can make a real impact on RAM requirements (otherwise, RAM requirements actually worsen, because the overlay loader module is added to the main program part).

Chapter 3

Using Overlays in Modula-2 Programs

The TopSpeed Modula-2 TechKit includes *two* overlay management programs. This chapter contains a brief description of the systems available for creating and using overlays. The **OVL** and **OVLXEXE** programs are discussed in separate sections.

Introduction to Overlays

The purpose of a program overlay system is to reduce the amount of RAM that an application program requires in order to run. This is accomplished by splitting the application program into a main part and two or more overlays. The application program then loads the overlays only when it needs to execute some code in them.

This really works well only if the following two conditions hold:

- each overlay has a big enough task so that the time to load the overlay is insignificant in comparison with the amount of work that the overlay has to do (otherwise, it is very slow)
- at least two of the overlays are big enough so that having only one of them at a time in memory can make a real impact on RAM requirements (otherwise, RAM requirements actually *worsen*, because the overlay loader module is added to the main program part).

Using the ovl Program

Please Note: This bit of software was designed and implemented for internal use by JPI. It is not user friendly and any mistakes you make will probably crash your machine hopelessly. It is, however, flexible and efficient.

There are two components:

- **OVL** is used to split an **.EXE** file into overlays.
- The **Overlay** module is used in the running program, to read in the overlays.

In order to give you maximum flexibility, **OVL.MOD**, the program which makes overlays, is provided in source form. As a first step, you should compile and link the **OVL** program itself.

The **OVL** program modifies your **.EXE** files to enable you to use overlays for the program. An **overlay** consists of a set of complete segments, which are not loaded initially, but which can be brought in by the call

Overlay.Load(n)

where **Load** is a predefined procedure in the **Overlay** module included in the TechKit. (The module also contains procedure **UnLoad**, for removing an overlay from memory.) You can have more than one overlay loaded at a time (if memory permits).

The parameter, **n**, is the number of the overlay to load, and must be in the range, 1 .. 9. This limit of 9 overlays can be changed. (The relevant parameters are in line 6 of **OVL.MOD** and line 54 of **OVERLAY.MOD**). The default limit was chosen because most applications will not require anywhere near 9 overlays.

After being loaded, all procedures in an overlay may be called as normal. **Overlay.Load** calls **Storage.HeapAllocate** to obtain the memory necessary to load-in the overlay.

Thus, the **Overlay** module makes it possible to move overlays into and out of your program.

To create the overlay(s), you need to do several things. Suppose your program is called **FRED**. The link command is

```
m2/l fred/m
```

as normal.

Now create a text file, **FRED.OVL**, with an entry for each segment or group you wish to put in an overlay. Each entry should be on a separate line. The format of each line

is a number from 1–9, followed by the group name (or segment name if there is no group name).

For example, suppose you want to overlay modules **FRED1** and **FRED2** into one overlay and **FRED3** into another. The **FRED.OVL** file would contain the following:

```
1 G_FRED1
1 G_FRED2
2 G_FRED3
```

The command to create the overlays and to modify **FRED.EXE** is

```
ovl fred
```

This reads the **FRED.OVL** file, and creates the appropriate overlay file(s).

The overlays will be in files **FRED.OV1**, **FRED.OV2** .. **FRED.OV9**.

Bear in mind that it is the *program's* responsibility to load the overlays. The map is modified to include only resident code, and the old map is copied to **FRED.OMP**.

Handling Small Segments and Other Worries

There is a slight flaw in the concept where segments are less than one paragraph (16 bytes). One consequence of this flaw is that the **OVL** program cannot distinguish the segment to which a segment fixup applies. Thus, if two segments in different overlays have the same paragraph, chaos will result.

Therefore, to be safe, all segments should be at least one paragraph. The **OVL** program issues a warning whenever this is a problem. If this happens you should enlarge the segment to at least 16 bytes by adding some extra variable declarations. Or, you can use **\$D** or **\$M** to group segments together.

Initialization code also tends to cause problems, since initialization code can only be used in the main, calling module, and can't be used in **.OVn** files. In this case, the simplest solution is to use explicit procedure calls for modules which are non-resident, and then execute these (after loading the overlay) from another (resident) module's initialization.

Linking Large Programs

The TopSpeed linker cannot link arbitrarily large programs. Normally this is not a problem because a program large enough to exceed the linker's capacity would probably be too large to run on a PC. However, through the use of overlays, this

assumption is violated. In such a case, the linker will report "Sorry: symbol table too large."

This problem will probably start to happen when the total code in the program (that is, the **.EXE** file size) is 300-800K bytes. The exact size depends strongly on the programming style. (Note that the TopSpeed Modula-2 system, which is about 300K bytes of code, links without any problem.)

Here are some tips on how to deal with this problem:

Use the Batch Linker

Use the batch version of the linker, with the **/1** switch. This has the effect of truncating global names to seven characters, which may produce name clashes. You will have to resolve these. The linker will report these as 'duplicate publics.' Note that a name can only clash with another name in the same module, since all public symbols are prefixed by the module name.

Structure Global Variables

Structure global variables into records. For example use:

```
VAR g =
  RECORD
    i, j, k : CARDINAL;
  END;
```

instead of

```
VAR i, j, k: CARDINAL;
```

This is because each global variable (whether exported or not) appears in the symbol table.

Reduce the Number of Segments

Use the **(*\$M codesegname*)** and **(*\$D datasegname*)** directives as much as possible to reduce the number of segments. This should also reduce the **.EXE** file size considerably.

Decrease the Number of Procedures

Avoid large numbers of very small procedures. For example, use one procedure to do the work of several by taking an extra parameter.

Suppress Libraries When Possible

Compile modules which do not require "smart linking" with the Suppress Libraries option (**/J**).

These measures should allow you to build very large programs, up to 1MB of code. However, at some stage you will have to resort to splitting your program into several separately linked programs — if for no other reason than that the turnaround time for program development would become unacceptable.

Typically, you would have a 'base' program, with commonly used routines. This program calls other programs to perform various application tasks. For the applications to call routines in the 'base' you will need to devise a method of vectoring the calls — for instance, by using a software interrupt. This would need to be written in assembler. To access global variables you should call a routine which returns a pointer to the global variable, which can then be used for future accesses to those variables. The global variables should be structured into records.

Of course, in OS/2 these problems should be a thing of the past...

Using the OVLEXE Program

The second overlay system allows the applications programmer to specify quickly and easily which program parts go into overlays. This is done by creating a small text file that is read in by the overlay generation program. The applications programmer also has complete control over the loading and unloading of overlays, and must take care that program parts are loaded before they are used. Although the overlay system has several utilities and features that make it easy for the programmer to accomplish and verify this, it is ultimately the programmer's responsibility to make sure that a program part is loaded before it is used.

The smallest indivisible program part is one linker group (or segment if not grouped). If you use the compiler defaults, all the data for a single module goes into one segment, and all the code for a single module goes into one group. Therefore you cannot put two procedures from the same module into two different overlays; but you *can* put the code from a module into an overlay, and the data from the same module into a different overlay (which will typically be the main program part).

Example

Assume that you have a program that does three main tasks.

No matter how many modules your total program uses, if the routines that do the three main tasks are in different modules that don't reference each other (although they may reference common modules) then you can split them out into different overlays. (You can split them out even if they do reference each other, but it makes for a more complicated example).

If your original main program is:

```
MODULE Main;
```



```
IMPORT mod1, mod2, mod3;
```

```
BEGIN
  mod1.DoIt;
  mod2.DoIt;
  mod3.DoIt;
END Main.
```

it now becomes:

```
MODULE Main;

IMPORT mod1, mod2, mod3, OvlMan;

BEGIN
  OvlMan.Load('mod1');
  mod1.DoIt;
  OvlMan.UnLoad('mod1');

  OvlMan.Load('mod2');
  mod2.DoIt;
  OvlMan.UnLoad('mod2');

  OvlMan.Load('mod3');
  mod3.DoIt;
  OvlMan.UnLoad('mod3');
END Main.
```

After you have compiled and linked your program using TopSpeed, you need to run it through the **OVLEXE** overlay-generation program. This program reorganizes your **MAIN.EXE** file so that your overlays are not in RAM by default, and may be loaded and unloaded under the control of the overlay module. All of the overlays will be contained in the **.EXE** file.

To do this, first create a text file named **MAIN.ODF**, in the same directory as **MAIN.EXE**. In **MAIN.ODF**, put the following line:

```
mod1 mod2 mod3
```

This will tell **OVLEXE** that you wish to create overlays from the modules **mod1** through **mod3**.

Now, enter the command

```
ovlexe main
```

OVLEXE will read **MAIN.MAP**, **MAIN.EXE**, and **MAIN.ODF**. It will then do the following:

- Rename **MAIN.EXE** to **MAIN.OEX** (for original **.EXE**).
- Create a new **MAIN.EXE**, having the overlays set up the way you requested.

See below if **OVLEXE** gives you any warnings.

Restrictions on the Use of Overlays

Any module with initialization code at the bottom cannot be placed into an overlay with this version of **OVLEXE**. **OVLEXE** will warn you if you have placed initialization code into an overlay. **OVLEXE** will *not* warn you if you have placed data that initialization code references into an overlay, but you have to explicitly do that, or it will wind up in the main program anyway.

Two linker segments or groups that have the same base address *must* be in the same overlay. Two segments can only have the same base address when they are adjacent in the original **.EXE** file, and the first one is less than sixteen bytes long. If you have chosen two different overlays for segments with the same base address, **OVLEXE** will automatically move both of them to the main program part, and issue a warning telling you it has done so. This will usually work, but if the second segment was quite long and you really wanted it in an overlay, you will have to manually pad the first segment. You can do this by adding dummy data to the first segment.

However, beware the tendency of the TopSpeed linker to leave out unreferenced data and procedures. The best way to increase data size is to add a dummy field at the end of a referenced record.

Another possibility is to use the \$D compiler directive option for the segment that contains fewer than 16 bytes of data or code.

OVLEXE Warnings

When you turn a module into an overlay, you are asserting to **OVLEXE** that the overlay will be loaded whenever you reference any code or data in it. **OVLEXE** may put more than just that module into the overlay — by default, it will place every module that is only referenced by one overlay into that overlay.

Also, **OVLEXE** makes *invisible* overlays — for example, by default, every module that is only referenced by named overlays will be turned into an invisible overlay. This invisible overlay will be loaded automatically whenever one of the named overlays that uses it is loaded, and will be unloaded automatically whenever all of the overlays that use it have been unloaded.

This is normally not a problem, but there can be cases where it is. Most of these cases relate to pointers. If the main program directly references a procedure or variable in a module that you didn't explicitly assign to an overlay, then that module will be assigned to the main program. However, if the main program *indirectly* references a procedure or variable in a module that you didn't explicitly assign to an overlay then that module may wind up in the overlay, because **OVLEXE** does not know about

indirect references. This can happen, for example, when a module that you *did* assign to the overlay loads a procedure variable in the main program with a procedure in the second overlay.

If the main program tries to access that procedure or variable (indirectly, through a pointer) when the overlay isn't loaded, there is a problem. Another, more insidious problem, occurs when the overlay is loaded, and a pointer is set up in the main program. The overlay may later be unloaded and then loaded again without resetting the pointer in the main program. Because the overlay is on the heap, there is no guarantee that it will be loaded in the same place the second time.

There are several considerations in deciding what goes into which overlay, and when each overlay should be loaded. Any segments/groups that you do not explicitly assign to overlays (using the `.ODF` file) will be implicitly assigned to overlays (or the main program part) by the `OVLEXE` program, so an understanding of the implicit assignment rules is essential.

Implicit Assignment Rules

Any segments that are not referenced at all, are referenced by the main program, or are referenced by *all* the named overlays, will be placed into the main program.

Any segments that are *only* referenced by one overlay will be assigned to that overlay.

Any segments that are referenced by two or more overlays, will be placed into an invisible overlay that will be loaded whenever one of the referencing overlays is loaded. These segments will be grouped, whenever possible, into common invisible overlays to reduce loading overhead.

Operation of OVLEXE

On the command line, `OVLEXE` is given the name of a program. `OVLEXE` then reads several files with the same name as the program but with different extensions. In particular, `OVLEXE` reads `<progrname>.EXE`, `<progrname>.MAP`, and `<progrname>.ODF`, and generates a new `.EXE` file.

`OVLEXE` keeps a copy of the original `.EXE` file in a file named `<progrname>.OEX`. When `OVLEXE` is invoked, if `<progrname>.EXE` has already been run through `OVLEXE`, it deletes it and reads the old `.OEX` file. IF `<progrname>.EXE` has *not* already been run through `OVLEXE`, `OVLEXE` deletes the old `.OEX` file (if any), and renames the `.EXE` file to `.OEX` before using it. In this way, the user can repeatedly

modify the **.ODF** file to try out different combinations without having to relink the program each time.

All the actual code (both main program and overlays) is stored in the **.EXE** file. Limits recorded in the file cause MS-DOS to ignore the overlays and only load up the main program, which then controls the loading of the overlays.

Options

Options are specified to **OVLEXE** with a **<programe>.ODF** file. This **.ODF** file contains lines with the following format:

<overlay name or MAIN> [<overlay or segment name>...]

All the words are considered to be either segment names or overlay names. The reserved word **MAIN** is the overlay name for the main program part. The first non-keyword on a line is *always* considered to be an overlay name. If no overlay of that name already exists, then one is created.

Overlays can be named anything (except **MAIN**), but, when a new overlay is created, **OVLEXE** automatically searches for segments or groups named **G_<name>**, **C_<name>**, and **D_<name>**. If any of these exist, they are automatically placed into the newly-created overlay. In this way, if you give overlays the same name as a Modula-2 module, you do not have to manually assign any segments to them. This first overlay name becomes the "currently selected overlay."

Subsequent non-reserved words on the same line are first assumed to be segment names. If they are segment names, then the named segments are placed into the currently selected overlay. If they are not valid segment names, then they are treated as overlay names, and become the "currently selected overlay."

The main program (overlay 0) can *only* access *named* overlays. All code that the main program uses will reside either in the main program, or in overlays that the main program can explicitly load (using their names).

Other overlays, however, can make use of *unnamed* (invisible) overlays. This happens automatically. Any segments that the main program does not reference directly, and that the **.ODF** file has not committed to a named overlay are considered "floating". If more than one overlay references such a floating segment, the segment will be placed in an invisible overlay. The **Overlay** module that is linked into the application program will automatically load in invisible overlays whenever visible overlays that use them are loaded, and will automatically unload them whenever all visible overlays that use them have been unloaded. Any module data that should have a lifetime of the entire program should be explicitly placed in the **MAIN** overlay.

The **MAIN** overlay (which is the resident part of the program) may be accessed as well. Segments may be explicitly assigned to **MAIN**, as with any other overlay, but there are two special cases:

1. If any segments are explicitly assigned to **MAIN**, then the *first* segment that is explicitly assigned to **MAIN** is special – it is considered to be the initialization segment. With TopSpeed, the first segment assigned to main should normally be **G_CODE**.
2. If no segments are explicitly assigned to **MAIN**, then **G_CODE** is implicitly assigned to **MAIN** and is considered to be the initialization segment.

OVLEXE checks the initialization segment. If it directly references code or data in any other overlays, a warning is issued. It is assumed that the initialization segment will be able to load the overlays before other main program segments that reference the overlays are invoked.

The OvlMan Module

The **OvlMan** module, which should be linked into your program (and should reside in the main program part) contains variables and routines that you can use to manage your overlays. You can accomplish the following things:

- Load overlays.
- Unload overlays.
- See if an overlay is loaded.
- See if the **OVLEXE** program has been run on the **.EXE** file yet or if all the code was loaded when the program ran.
- Install your own debugging routine that **OvlMan** will call whenever it loads or unloads overlays (including invisible overlays). This routine will be passed the overlay number and load segment.

See **OvlMan.DEF** for complete details of the interface to the **OvlMan** module.

OVLMAP and OVLXREF

OVLMAP and **OVLXREF** are programs that operate similarly to **OVLEXE**. They both perform the following actions:

- If a **.EXE** file exists that hasn't been passed through **OVLEXE**, they delete any **.OEX** files and rename the **.EXE** file to **.OEX**.
- They read in the **.OEX** file, the **.MAP** file, and the **.ODF** file.

After setting up like this, they do different things. **OVLMAP** generates map files for all the overlays. These files have extensions **.M00** through **.Mxx**, where **xx** is the highest overlay number. **.M00** shows the segments and labels that are in the main program part; **.M01** to **.Mxx** show all the segments and labels that wound up in the overlays. There is one file for each overlay, both named and invisible. These map files are adjusted to reflect offsets from the start of the overlays. The **ATRON** **AT-PROBE** debugger (and probably some others) can accept these files and also an offset to where the overlay has been loaded.

OVLXREF generates a file named **<programe>.XRF**. This file shows all the references between the various segments in the various overlays so that the application programmer can verify that overlays are not referenced any more often than they need to be.

Debugging Overlays

You can symbolically debug overlays fairly easily if your debugger allows you to "offset" symbol definitions that you give it. To do this, make the **UserDebug** routine in **Overlay** point to a procedure that will output the name of the overlay map file and the segment offset for that overlay to a disk file. This output will be in a format that your debugger can read. Then, when your program is loaded, have it load up the overlays to be debugged, and either manually or automatically breakpoint to the debugger. Then read in the map file descriptor with the debugger, and return control to the main program. (Current versions of the **TopSpeed VID Debugger** do not have this capability.)

Of course, in most cases, it will be easier to debug modules as part of a *non-overlaid* program *before* moving them into overlays.

More Technical Data

This section contains some technical information about how overlays are represented. This section is *not* required in order to use the **OVLEXE** package.

The **.EXE** file contains *all* the overlays, After the main part of the program, and a time/date code and a module directory *in front of* the main part of the program. Because of limits recorded in the **.EXE** file header, the DOS loader will not attempt to load any overlays. We can selectively load them after the program has started execution. The time/date code and module directory is loaded immediately, however. It is made to be a part of the main program and will be located at **PSP:100h**.

This module directory is defined as a Modula-2 record. It contains these items:

- A “key” so that the loader can determine if the record is valid, or if it really exists at all.
- A version number for the overlay generator/loader. If the loader is a different version number than the overlay generator, it will print an error message and stop.
- A pointer to itself in the file. When the file is opened to read an overlay, the record is reread from this location in the file, and compared with the original record. If they are not the same, we didn't find the right file, so we print an error message and stop.
- The number of data segments in the file.
- The name of the program file. This is used by the loader to relocate the **.EXE** file for loading overlays.
- The time and date when **OVLEXE** was run on the file. This is so that the loader module can verify it found the correct file and not an earlier or later version that happened to be in the search path.
- Overlay names and associated overlay numbers. These are arranged in one string (the overlay numbers are all > 0). The string can be quickly searched using **Str.Pos**. The format is:

<single byte number><space><string name><space> etc.

The user requests an overlay by passing a string to the loader or unloader. For example,

```
OvlMan.Load ('MyOverlay')
```

This string is copied into another string, converted to uppercase, padded with blanks, and then passed to **Pos**. The overlay number is at one less than the index returned from **Pos**.

The module directory record also contains:

- The total number of overlays in the file (excluding main program)
- A record for each overlay (visible and invisible) containing:
 1. The offset in the **.EXE** file where the code starts
 2. The number of segments to load
 3. The offset in the **.EXE** file where the internal relocation chain starts (modification of vectors within this overlay) or 0 for none.
 4. The offset in the **.EXE** file where the external relocation chain starts (modification of vectors within other overlays) or 0 for none.
 5. A set of overlays that this overlay implicitly uses (named overlays only)

After the main program (and not loaded by DOS), are the actual code segments and the relocation lists.

The relocation lists (which are pointed to by the overlay directory) are relocation item lists which are singly-linked (no back links) but doubly-threaded. There is one thread for internal relocations and another thread for external relocations.

Each relocation is only in the file once, although it may be linked into two chains. The relocations always go forward in the file, and the loader reads fairly large chunks at a time so that it doesn't make one FIO call for each relocation.

The relocation fixups consist of:

- Number of overlay to be patched
- Number of overlay to point the patch at (target)
- Segment and offset within the patch overlay
- Segment within the target overlay

A full definition of these records is in **OvldDirectory.DEF**.

Before discussing the Toplevel assembly language in greater detail, this chapter contains a brief discussion of the architecture and instruction set for the 80286. The Toplevel assembly language is discussed in detail in the following chapter, with examples in the chapter following that. Certain obscurities are not described, those to be found in, for example, Intel's 80286 and 80287 Programmer's Reference Manual.

Registers

There are 14 16-bit registers, as shown in Table 4-1. These 16-bit registers are split into three main groups: the data group, the pointer and index group, and the segment register group.

The data group consists of four registers, each 16-bit wide. These are ax, bx, cx, and dx. These four registers can be used either as single 16-bit registers, or as two 8-bit registers - referenced as the lower half (l) and the upper half (h) - labelled al, ah, bl, bh, cl, ch, dl, dh.

The pointer and index group, consisting of sp, bp, si, and di, are only used as 16-bit registers.

The 80286/287 provides four 16-bit segment registers, the CS, DS, SS, and ES registers (code, data, stack, and extra segment registers). On the 80286, the

Chapter 4

8088/8087 Instruction Sets

Table 4-1: Registers in 8088/8087. The registers are split into three main groups: the data group, the pointer and index group, and the segment register group.

Register	Size	Group
ax	16-bit	Data
bx	16-bit	Data
cx	16-bit	Data
dx	16-bit	Data
si	16-bit	Pointer/Index
di	16-bit	Pointer/Index
sp	16-bit	Pointer/Index
bp	16-bit	Pointer/Index
ip	16-bit	Pointer/Index
cs	16-bit	Segment
ds	16-bit	Segment
ss	16-bit	Segment
es	16-bit	Segment

Before discussing the TopSpeed assembly language in greater detail, this chapter contains a brief discussion of the architecture and instruction set for the 8086/8087. The TopSpeed assembly language is discussed in detail in the following chapter, with examples in the chapter following that. Certain obscurities are not described; these can be found in, for example, Intel's *80286 and 80287 Programmer's Reference Manual*.

Registers

There are 14 16-bit registers, as shown in Table 4-1. These fourteen registers are split into three main groups: the data group, the pointer and index group, and the segment register group.

The data group consists of four registers, each sixteen-bits wide. These are **ax**, **bx**, **cx**, and **dx**. These four registers can be used either as single 16-bit registers, or as two 8-bit registers – referenced as the lower half (l) and the upper half (h) – labelled **ah**, **al**, **bh**, **bl**, **ch**, **cl**, **dh**, **dl**.

The pointer and index group, consisting of **sp**, **bp**, **ip**, **si** and **di**, are only used as 16-bit registers.

The 8086/8088 provides four 16-bit segment registers, for the CODE, DATA, STACK, and EXTRA segments (**cs**, **ds**, **ss**, and **es**, respectively). On the 8086/88 (and

on the 80286/386 in real mode), the segment registers are combined with other (index) 16-bit values to obtain a 20-bit wide address.

In protected mode on the 80286/386, the segment registers instead contain **selectors**, which are used by the CPU as indexes into **descriptor tables**. These tables contain the actual address component (along with other information, such as access rights). This address component is combined with other (index) 16-bit (possibly 32-bit on the 80386) values to obtain a 24-bit wide address (32-bit wide on the 80386).

Table 4-1

8086 Registers

Short Form	Pronounced
cs	code segment register
ss	stack segment register
ds	data segment register
es	extra segment register
ax	ax register (the "accumulator")
bx	bx register
cx	cx register
dx	dx register
sp	stack pointer
bp	base pointer
si	source index register
di	destination index register
ip	instruction pointer
fl	flag registers

There are, in addition, nine 1-bit flag registers grouped together within a 16-bit wide flag register known as **fl**.

- OF overflow flag — an arithmetic overflow has occurred, that is a result has exceeded the capacity of a destination register/address, and the significant bit has been lost.
- DF direction flag — determines direction of data transfer; when set, string operations decrement SI and DI giving right-to-left data transfer, when not set (0), SI and DI are incremented.
- IF interrupt enable flag (hardware interrupts can occur if IF=1)
- TF trap flag — when set this causes program execution to single step one instruction at a time (for use by software debuggers)

- SF sign flag — this flag is set if the most significant bit is set (since negative numbers are represented in a two's complement form).
- ZF zero flag — set when the result of an operation is zero.
- AF auxiliary flag — if set, a low to high nibble carry or a high to low nibble borrow has occurred (not usually useful)
- PF parity flag — if set, the result of an operation resulted in an even number of one's in the word (not usually useful)
- CF carry flag — set if there has been a carry or borrow operation to the most significant bit.

These flag registers are stored in the format within the flag register shown in Table 4-2.

Table 4-2

Layout of Flags Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	OF	DF	IF	TF	SF	ZF	xx	AF	xx	PF	xx	CF

(xx means not used)

OF, SF, ZF, AF, PF and CF are set by arithmetic/logical operations.

OF is set if there is a signed overflow

ZF is set if the result is zero

CF is set if there is an unsigned overflow

Note that **inc** and **dec** do *not*, however, change CF!

Data transfer instructions such as **mov**, **push**, **pop** do not change the flags. For further details on when flags are set you will have to consult the Intel documentation, although this should not normally be necessary.

Memory Addressing

cs, **ss**, **ds** and **es** are known as segment registers. Their only use is to act as base registers for memory addresses, and every memory operand is based on one of these segment registers. The byte to be accessed has an address that is calculated by the $16 \times \text{segment-register} + \langle \text{a 16-bit offset} \rangle$ formula. This gives a 20-bit memory address space (which is 1 megabyte of addressable locations). Memory addresses are represented in the form **segment:[offset]**.

When addressing any byte, be it code or data, a segment register is used. The register used is dictated by the assembler, which, in turn, is directed by the programmer.

Instructions

Instructions are fetched from the location **cs:[ip]**.

After the instruction is fetched (but before it is executed) the **ip** register is updated to point to the next instruction. Apart from this automatic updating, the **cs** and **ip** registers can only be updated by **jump/call/ret/int/iret** instructions.

An 8086 instruction is specified by an opcode followed by an optional specifier, followed by 0,1 or 2 operands. That is,

opcode specifier operand1, operand2

For example:

```
mov byte es:[bx],1
```

The specifier is used to indicate the data size where more than one data size is possible and the size is not implied by a register operand.

byte	= 1 byte
word	= 2 bytes
dword	= 4 bytes
qword	= 8 bytes (floating point only)
tbyte	= 10 bytes (floating point only)
near	= 2 bytes (relates to jump/call/return instructions)
far	= 4 bytes (relates to jump/call/return instructions)

Operands

An operand may be

- a general register (one of **ax, bx, cx, dx, sp, bp, si, di, al, bl, cl, dl, ah, bh, ch, dh**)
- a segment register (one of **es, ss, ds, es**)
- a constant (either a number or a label)

- a memory operand

In the following description these types are known as **r**, **s**, **c**, and **m** respectively. The possible combinations of operands for an instruction are shown in Table 4-3:

Table 4-3

8086 Operands

Abbreviation	Description
	no operand required
r	general register
s	segment register
c	constant
m	memory
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

For example:

```

mov ax,bx      (* Register to register move -- contents of bx to ax *)
mov ax,aWord   (* Contents to register -- move contents of aWord to ax *)
mov ax,[bx]    (* Contents of Location to reg. -- move contents of *)
               (* location specified by bx to ax *)
mov ax,20      (* Move value 20 to register ax *)
mov ax,[20]    (* Contents of location to reg. -- move contents of *)
               (* location of offset 20 to ax *)
mov word[20],20 (* Constant value 20 into offset 20. *)
               (* Note specifier for data size. *)
               (* Neither operand defines the size *)

```

The other possibilities (such as memory, memory) are not available.

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 a segment register followed by a colon.

segment : [**offset1**] [**offset2**] [**offset3**] ...

If no segment override is specified, then **ds** is used for the segment, unless one of the offsets is the **bp** register in which case **ss** is used.

The offsets may be labels, numbers, or index registers. The index registers are **bx**, **bp**, **si**, **di**

The eight legal combinations of index registers are: [**bx**], [**bp**], [**si**], [**di**], [**bx**][**si**], [**bx**][**di**], [**bp**][**si**], and [**bp**][**di**].

Opcode Descriptions

In Table 4-4, which describes each opcode in turn:

- **op1** and **op2** stand for the first and second operands of the instruction.
- Where the specifiers are given as **near/far**, the **cs** segment register is involved only in the **far** case.
- The notation **a:b:c...** is used to indicate multi-word operands. The components are stored in decreasing locations in memory. For example, **cs:ip**, where **cs** is high order and **ip** is low order.

The following notations are used in the description:

x < y	means $x := x * 2^y$
x > y	means $x := x \text{ DIV } 2^y$
push(x)	means $sp := sp - 2; ss:[sp] := x$
pop(x)	means $x := ss:[sp]; sp := sp + 2$
next(x, i)	means if DF (direction flag) then $x := x - i$ else $x := x + i$
rol(x, y)	means $x := (x \ll y) + (x \gg \text{size}(x) - y)$
ror(x, y)	means $x := (x \gg y) + (x \ll \text{size}(x) - y)$
sar(x, y)	means shift x right y times, preserving sign of x
? :=	means that the result is discarded

Instructions whose results are discarded — as in the last item in the preceding list — are generally executed for their effects on the flags.

Table 4-4

8086 Opcodes

Opcode	Possible Specifiers	Possible Operand Combinations	Description in Modula-2 like terms
aaa			not described here
aad			not described here
aam			not described here
aas			not described here
adc	byte/word	rr/mr/rm/rc/mc	op1 := op1 + op2 + CF
add	byte/word	rr/mr/rm/rc/mc	op1 := op1 + op2
and	byte/word	rr/mr/rm/rc/mc	op1 := op1 * op2 (as bitsets)
call	near/far	c	push cs; push ip; cs:ip := c
cbw			ax := integer(shortint(al))
clc			CF := 0
cld			DF := 0 (normal, incrementing)
cli			IF := 0
cmc			CF := ~CF
cmp	byte/word	rr/mr/rm/rc/mc	? := op1 - op2
cmpsb	byte		? := ds:[si]-es:[di]; next(di,1); next(si,1)
cmpsw	word		? := ds:[si]-es:[di]; next(di,2); next(si,2)
cwd			dx:ax := longint(integer(ax))
daa			not described here
das			not described here
dec	byte/word	r/m	op1 := op1 - 1
div	byte	r/m	al:=ax div op1; ah := ax mod op1 (unsigned)
div	word	r/m	ax:=dx:ax div op1; dx:=dx:ax mod op1 (unsigned)
hlt			not described here
idiv	byte	r/m	al := ax div op1; ah := ax mod op1 (signed)
idiv	word	r/m	ax:=dx:ax div op1; dx:=dx:ax mod op1 (signed)
imul	byte	r/m	ax := al * op1 (signed)
imul	word	r/m	dx:ax := ax * op1 (signed)
in	byte/word	rc/rr	op1 := in(op2) (op1 is al/ax; op2 is const/dx)
inc	byte/word	r/m	op1 := op1 + 1
int		c	push FL:cs:ip; cs:ip := 0:[4*c]; IF:=0
into			if OF then push FL:cs:ip; cs:ip:=0:[4*4]; IF:=0
iret			pop ip; pop cs; pop FL;
ja	near/far	c	if ~CF AND ~ZF then cs:ip := op1
jbe	near/far	c	if CF OR ZF then cs:ip := op1
jc	near/far	c	if CF then cs:ip := op1
jcxz		c	if cx=0 then ip := op1 (all flags unchanged)
jg	near/far	c	if ~ZF AND SF=OF cs:ip := op1
jge	near/far	c	if SF=OF then cs:ip := op1
jl	near/far	c	if SF#OF then cs:ip := op1
jle	near/far	c	if ZF OR SF#OF then cs:ip := op1
jmp	near/far	c	cs:ip := c

— continued —

Table 4-4 (continued)

Opcode	Possible Specifiers	Possible Operand Combinations	Description in Modula-2 like terms
jnc	near/far	c	if ~CF then cs:ip := op1
jno	near/far	c	if ~OF then cs:ip := op1
jns	near/far	c	if ~SF then cs:ip := op1
jnz	near/far	c	if ~ZF then cs:ip := op1
jo	near/far	c	if OF then cs:ip := op1
jp	near/far	c	if PF then cs:ip := op1
jpo	near/far	c	if ~PF then cs:ip := op1
js	near/far	c	if SF then cs:ip := op1
jz	near/far	c	if ZF then cs:ip := op1
lahf	byte		ah := shortcard(FL)
lds		rm	ds:op1 := op2
lea		rm	op1 := ADR(op2)
les		rm	es:op1 := op2
lock			not described here
lodsb			al := ds:[si]; next(si,1)
lodsw			ax := ds:[si]; next(si,2)
loop		c	cx := cx-1; if cx#0 then ip := op1
loopnz		c	cx := cx-1; if cx#0 and ~ZF then ip := op1
loopz		c	cx := cx-1; if cx#0 and ZF then ip := op1
mov	byte/word	rr/rc/rm/mr/mc	op1 := op2
mov	word	sr/sm/ms/rs	op1 := op2
movsb	byte		es:[di] := ds:[si]; next(di,1); next(si,1)
movsw	word		es:[di] := ds:[si]; next(di,2); next(si,2)
mul	byte	r/m	ax := al * op1 (unsigned)
mul	word	r/m	dx:ax := ax*op1 (unsigned)
neg	byte/word	r/m	op1 := 0 - op1
nop			(*skip*)
not	byte/word	r/m	op1 := bitset0..15 - op1
or	byte/word	rr/mr/rm/rc/mc	1 := op1 + op2 (as bitsets)
out	byte/word	cr/r	out(op1,op2) (op1 is const/dx; op2 is al/ax)
pop	word	s/r/m	pop op1
popf	word		pop FL
push	word	s/r/m	push op1
pushf	word		push FL
rcl	byte/word	rr/rc	rcl(op1:CF, op2) (op2 is cl/1)
rcr	byte/word	rr/rc	ror(op1:CF, op2) (op2 is cl/1)
rep			repeat following string instruction cx times and while ZF for scasb/scasw/cmpps/cmpsw
repnz			repeat following string instruction cx times or until ZF for scasb/scasw/cmpps/cmpsw
ret	near/far	c	pop ip; pop cs; sp:=sp+op1
rol	byte/word	rr/rc	rol(op1, op2) (op2 is cl/1)
ror	byte/word	rr/rc	ror(op1, op2) (op2 is cl/1)

— continued —

Table 4-4 (continued)

Opcode	Possible Specifiers	Possible Operand Combinations	Description in Modula-2 like terms
sahf	byte		FL := (FL DIV 256)*256 + cardinal(ah)
sar	byte/word rr/rc		sar(op1, op2) (op2 is cl/1)
sbb	byte/word rr/mr/rm/rc/mc		op1 := op1 - (op2 + CF)
scasb	byte		? := al - es:[di]; next(di,1)
scasw	word		? := ax - es:[di]; next(di,2)
shl	byte/word rr/rc		op1 := op1 << op2 (op2 is cl/1)
shr	byte/word rr/rc		op1 := op1 >> op2 (op2 is cl/1)
stc			CF := 1
std			DF := 1
sti			IF := 1
stosb	byte		es:[di] := al; next(di,1)
stosw	word		es:[di] := ax; next(di,2)
sub	byte/word rr/mr/rm/rc/mc		op1 := op1 - op2
test	byte/word rr/mr/rm/rc/mc		? := op1 * op2 (as bitsets)
wait			not described here
xchg	byte/word rr/mr/rm		temp := op1; op1 := op2; op2 := temp
xlat			al := ds:[bx+cardinal(al)]
xor	byte/word rr/mr/rm/rc/mc		op1 := op1 / op2 (as bitsets)

After the instruction,

`cmp op1, op2`

the following interpretations of the jump instructions are valid :

`je lab` if op1 = op2 then goto lab (jump equal)
`jne lab` if op1 # op2 then goto lab (jump not equal)

For cardinal arithmetic:

`jb lab` if op1 < op2 then goto lab (jump below)
`ja lab` if op1 > op2 then goto lab (jump above)
`jbe lab` if op1 <= op2 then goto lab (jump below or equal)
`jae lab` if op1 >= op2 then goto lab (jump above or equal)

For integer arithmetic:

`j1 lab` if op1 < op2 then goto lab (jump less)
`jg lab` if op1 > op2 then goto lab (jump greater)
`jle lab` if op1 <= op2 then goto lab (jump less or equal)
`jge lab` if op1 >= op2 then goto lab (jump greater or equal)

Floating Point (8087) Instructions

The 8088 is rather limited when it comes to mathematical operations on anything other than small integers. If you want to carry out operations that have more than four decimal digits, you would normally be stuck. But fear not, the answer lies in the 8087 co-processor (or if you do not have one, in an emulation library).

The 8087 has a stack-based architecture with eight stack-based 10-byte registers. These provide an accuracy of 18 digits (more than enough to, say, calculate the time saved using the TopSpeed package!) or 16 digits of accuracy if you use floating point numbers.

These are addressed relative to an internal pointer **ST** which is updated by some operations. The 8087 registers are denoted by **st(0)**, **st(1)** ... **st(7)**. **st(0)** and **st(1)** have special status.

In addition, the 8087 has the 16-bit registers listed in Table 4-5:

Table 4-5

8087 Registers

Short Form	Pronounced
CW	Control word
SW	Status word
TW	Tag word
IPL	Instruction Pointer Low (for the 8087)
IPH	Instruction Pointer High
DPL	Data Pointer Low
DPH	Data Pointer High

These are of little interest normally, except that the status word has to be transferred to the 8086 flag register to determine the result of 8087 comparison instructions.

The possible combinations of operands for a floating point instruction are shown in Table 4-6:

Table 4-6

Floating Point Operand Combinations

Abbreviation	Description
	no operand required
t	stack top (st(0))
r	any floating point register (st(0), st(1), st(2) ... st(7))
m	memory
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)

For the floating point instructions shown in Table 4-7, the following notation is used:

- **push** means $ST := ST - 1$
- **pop** means $ST := ST + 1$

Table 4-7

8087 Opcodes

Opcode	Possible Specifiers	Possible Operand Combinations	Description in Modula-2 like terms
f2xm1		t	op1 := 2**op1 - 1
fabs		t	op1 := ABS(op1)
faddp		rt	op1 := op1 + op2; pop
fadd	dw/qw	tr/rt/tm	op1 := op1 + op2
fbld		tm	push; op1 := op2 (op2 is packed decimal)
fbstp		mt	op1 := op2; pop (op1 is packed decimal)
fchs		t	op1 := -op1
fclex			not described here
fcom	dw/qw	tr/tm	? := op1 - op2
fcomp	dw/qw	tr/tm	? := op1 - op2; pop
fcompp		2t	? := op1 - op2; ST:=ST+2
fdecstp			push
fdivp		rt	op1 := op1 / op2; pop
fdiv	dw/qw	tr/rt/tm	op1 := op1 / op2
fdivrp		rt	op1 := op2 / op1; pop

— continued —

Table 4-7 (continued)

Opcode	Possible Specifiers	Possible Operand Combinations	Description in Modula-2 like terms
fdivr	dw/qw	tr/rt/tm	op1 := op2 / op1
ffree		r	not described here
fiadd	word/dw	tm	op1 := op1 + op2 (op2 is integer)
ficom	word/dw	tm	? := op1 - op2 (op2 is integer)
ficomp	word/dw	tm	? := op1 - op2; pop (op2 is integer)
fdiv	word/dw	tm	op1 := op1 / op2 (op2 is integer)
fdivr	word/dw	tm	op1 := op2 / op1 (op2 is integer)
fild	word/dw/qw	tm	push; op1 := op2 (op2 is integer)
fmul	word/dw	tm	op1 := op1 * op2 (op2 is integer)
fincstp			ST:=ST + 1
fini			initialize 8087
fist	word/dw	mt	op1 := op2 (op1 is integer)
fistp	word/dw/qw	mt	op1 := op2; pop (op1 is integer)
fisub	word/dw	tm	op1 := op1 - op2 (op2 is integer)
fisubr	word/dw	tm	op1 := op2 - op1 (op2 is integer)
fld	dw/qw/tb	tr/tm	temp := op2; push; op1 := temp
fldcw		m	CW := op1
fldenv		m	DPH:DPL:IPH:IPL:TW:SW:CW := op1
fldl		t	push; op1 := push 1.0
fldl2e		t	push; op1 := log base 2 of e
fldl2t		t	push; op1 := log base 2 of ten
fldlg2		t	push; op1 := log base 10 of 2
fldln2		t	push; op1 := log base e of 2
fldpi		t	push; op1 := 3.14159... (Pi)
fldz		t	push; op1 := 0.0
fmulp		rt	op1 := op1 * op2; pop
fmul	dw/qw	tr/rt/tm	op1 := op1 * op2
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	whole 8087 state := op1 (94 bytes)
fsave		m	op1 := whole 8087 state (94 bytes)
fscale		2t	op1 := op1 * (2**op2)
fsqrt		t	op1 := square_root(op1)
fst	dw/qw	rt/mt	op1 := op2
fstcw		m	op1 := CW
fstenv		m	op1 := DPH:DPL:IPH:IPL:TW:SW:CW
fstp	dw/qw/tb	rt/mt	op1 := op2; pop
fstsw		m	op1 := SW
fsubp		rt	op1 := op1 - op2; pop
fsub	dw/qw	tr/rt/tm	op1 := op1 - op2
fsubrp		rt	op1 := op2 - op1; pop

— continued —

Table 4-7 (continued)

Opcode	Possible Specifiers	Possible Operand Combinations	Description in Modula-2 like terms
fsubr	dw/qw	tr/rt/tm	$op1 := op2 - op1$
ftst		t	$? := ST - 0.0$
fwait			wait for co-processor to complete operation
fxam		t	not described here
fxch		tr/rt	$temp := op1; op1 := op2; op2 := temp$
fextract		t	push; $st(0) * (2^{st(1)}) := st(1)$ ($1 \leq st(0) < 2$)
fyl2x		t2	$st(1) := st(1) * \logbase2(st(0)); pop$
fyl2xp1		t2	$st(1) := st(1) * \logbase2(st(0)+1.0); pop$

A simple example to add two integers using the 8087 instructions would be:

MODULE ftest

segment C_CODE(CODE,28H); group G_CODE(C_CODE); select C_CODE;

```

entry:      jmp prog
            Number1 : dw 4          (* define one-word integers *)
            Number2 : dw 3
            Answer  : dw 0

prog:       finit                    (* initialize the 8087 *)
            fiild word st(0), [Number1] (* load int onto top of stack *)
            fiadd word st(0), [Number2] (* integer add *)
            fistp word [Answer], st(0)  (* integer store into Answer *)
            fwait                    (* allow 8087 to finish *)
            int      20H

end

```

Note the syntax used, and the specifiers required. To produce a program use the following:

```

m2asm ftest.a
link ftest
exe2bin ftest ftest.com

```

The results of the assembler can be seen if DEBUG is used:

```

-u
2ABC:0100 EB06      JMP      0108
2ABC:0102 0400      ADD      AL,00
2ABC:0104 0300      ADD      AX,[BX+SI]
2ABC:0106 0000      ADD      [BX+SI],AL
2ABC:0108 9B        WAIT
2ABC:0109 DBE3      FINIT
2ABC:010B 9B        WAIT
2ABC:010C DF060200  FILD      WORD PTR [0002]
2ABC:0110 9B        WAIT
2ABC:0111 DE060400  FIADD      WORD PTR [0004]
2ABC:0115 9B        WAIT
2ABC:0116 DF1E0600  FISTP     WORD PTR [0006]
2ABC:011A CD20      INT      20
2ABC:011C CA008A    RETF      8A00

```


2ABC:011F 44

INC

SP

Note the WAIT states inserted by the assembler to compensate for the 8087's computation time.

If **foption 2** is used, the WAIT states are suppressed.

After the sequence of instructions:

```
fcom op1,op2; fstsw [temp]; fwait; mov ax,[temp]; sahf
```

the following interpretations of the jump instructions are valid :

jb lab	if op1 < op2 then goto lab	(jump below)
jbe lab	if op1 <= op2 then goto lab	(jump below or equal)
ja lab	if op1 > op2 then goto lab	(jump above)
jae lab	if op1 >= op2 then goto lab	(jump above or equal)
je lab	if op1 = op2 then goto lab	(jump equal)
jne lab	if op1 # op2 then goto lab	(jump not equal)

Chapter 5

The TopSpeed Assembly Language

The TechKit includes an assembler, **M2ASM.EXE**, which you can use to assemble source files into **.OBJ** form. You can assemble files such as **ORDER.A** (which is included with the TechKit, and which is described elsewhere) or any of the system modules that are coded in assembler (such as **ASMLIB.A** and **SYSTEM.A**). You can even change and reassemble the run-time library, **RTL.A**. See the preceding chapter for a brief description of the assembly language and the instruction set for the 8086/8087 architecture.

The assembler is designed to be simple, fast, and to support the creation of libraries, just like TopSpeed Modula-2. To accomplish this, the assembly language differs from "standard" 8086 assembler in the following ways:

- Lexical structure is similar to that for Modula-2 — case is significant in identifiers, comments are included within **(* and *)**, and hex constants must be uppercase
- A **;** can be used to separate multiple statements on a line, and a newline is equivalent to **;**
- There is only one kind of label, and there are no data types — specifiers are used instead. Thus, **near**, **far**, **byte**, **word** can be used to specify the data type of an instruction
- Memory operands and segment overrides must always be explicitly specified
- No macros or other messy, complicated features are used

This section contains a brief description of the assembly language for the TopSpeed assembler.

The assembler achieves its high speed by translating the source in a single pass. This means that any variables and definitions must be defined before use. Otherwise the assembler can only assume that they are within the same segment.

How to Call the Assembler

The assembler can be invoked in two ways: from the command line or from within the environment. The **make** utility will invoke the assembler as needed, regardless of whether the **make** was started from the command line or from within the environment.

To assemble the **RTL.A** file (for example) from the command line, type

```
m2asm rtl
```

Any assembly errors will be written to the standard output.

To assemble from within the environment, select **Assemble** from the menu (or use the **Alt|A** shortcut), and specify that you wish to assemble **RTL**. Assembly errors will be reported in the same way that compiler errors are: The error editor window is invoked, and the user can move to each error to make corrections.

Program Style

When using the TopSpeed assembler, programs should follow the general layout shown in the next listing:

```
module name
code segment declaration and selection
main program
data segment declaration and selection
data declarations
initcode segment declaration (* optional *)
initialization code          (* optional *)
```


Segments

Any program consists of at least one segment, that is a code segment for the executable code. Often, a stack segment for the stack, and a data segment for data definitions are setup. Setting up each of these segments is the responsibility of the programmer, and should conform to the following declarative syntax:

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

For example,

```
segment C_FOO(FCODE,28H)
```

declares segment **C_FOO** in class **FCODE**, with an align/combine code of **28H**.

Similarly, the declaration of a group should conform to the following format:

```
group <group name> (<class>)
```

For example,

```
group G_FOO(C_FOO,S_FOO,K_FOO)
```

declares group **G_FOO**, containing segments **C_FOO**, **S_FOO**, and **K_FOO**.

The number in the segment definition determines the way in which bytes will be aligned and refers to the align/combination byte output to the object file. These are the same as the numbers output by the linker if you specify a detailed segment map.

The normal values to use are:

28H - byte aligned segment

48H - word aligned segment

68H - paragraph aligned segment

74H - stack segment

Byte aligned segments can start on any byte. Word aligned segments will start at an even address. Paragraph aligned segments start at an address which is a multiple of 16.

Example: Setting Up a Code Segment

For example, to declare and select a byte aligned code segment called **C_PROG** in a class named **CCODE** belonging to a group **G_PROG** the TopSpeed assembler directive would be of the form:

```
segment C_PROG(CCODE,28H)
group G_PROG(C_PROG)
select C_PROG
```

This is written into the very small (and very simple) example program below:

```
MODULE example

segment C_CODE(CODE,28H); group G_CODE(C_CODE); select C_CODE;

LetterJ : db 'J'

public example$main:    mov ax,cs: [LetterJ]    /* public Far procedure */
                        ret far 0                /* returns from a far call */
end
```

Example: Setting Up a Data Segment

A data segment would have declarations of the same form as the code segment definitions. However, the class name is dependent upon the contents of the segment.

If a data segment contains initialized variables, the segment should have class name **DATA**; otherwise it should have class name **M_DATA**. Segments in the latter class are initialized to zero if you use the **\$Z+** compiler option. Segments in the **DATA** class are *not* affected by this directive.

Tokens

The tokens that are applicable to the assembler are:

Generic Tokens

```
string    (* as in Modula-2 *)
number    (* as in Modula-2 *)
name      (* as in Modula-2 *)
```


register (*ax cx dx bx sp bp si di al cl dl bl ah ch dh bh
es cs ss ds *)

Keywords

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

Operator Precedence

The order in which the precedence of the operators is determined is as follows:

power2 (* highest precedence *)

log2

/ % *

+ -

:

~

&

|

seg (* lowest precedence *)

Operators of equal precedence associate to the left.

Syntax

The syntax for the assembly language is presented in this section. The notation used is as follows:

read '::=' as 'is defined to be',

read '|' as 'or alternatively'.

Comments are used to indicate the meaning of the construct; a period ('.') terminates the production.

name . (* normal label *)

- **<specifier> ::=** | (* default *)
 - near** |
 - far** |
 - byte** |
 - word** |
 - dword** |
 - qword** |
 - tbyte** .
- **<byteexprs> ::= <byteexp> | <byteexprs> ',' <byteexp> .**
- **<byteexp> ::= string** | (* outputs a string *)
 - <exp> .** (* outputs a byte *)
- **<wordexprs> ::= <wordexp> | <wordexprs> ',' <wordexp> .**
- **<wordexp> ::= <exp> .** (* outputs a word *)
- **<operand-llst> ::=** | (* no operands *)
 - <operand>** | (* one operand *)
 - <operand> ',' <operand> .** (* two operands *)
- **<operand> ::=**
 - register** | (* register operand *)
 - <exp>** | (* immediate operand *)
 - register ':' <addr-llst>** | (* segment override *)
 - <addr-llst>** | (* memory operand *)
 - st (<exp1>)** | (* 8087 register *)
 - seg <exp>** . (* segment of <exp> *)
- **<addr-llst> ::= <addr> | <addr-llst> <addr> .**
- **<addr> ::= '[' <exp> ']'** | (* displacement *)

'I' register 'J' . (index register *)*

□ *<exp> ::=*

<i>number</i>	
<i>name</i>	
<i>~' <exp></i>	(* bitwise NOT *)
<i>'.' <exp></i>	
<i><exp> '*' <exp></i>	
<i><exp> '/' <exp></i>	(* <exp> DIV <exp> *)
<i><exp> '%' <exp></i>	(* <exp> MOD <exp> *)
<i><exp> '+' <exp></i>	
<i><exp> '-' <exp></i>	
<i><exp> '&' <exp></i>	(* bitwise logical AND *)
<i><exp> ' ' <exp></i>	(* bitwise logical OR *)
<i>'(' <exp> ')'</i>	
<i>power2 <exp></i>	(* 2**<exp> *)
<i>log2 <exp></i>	. (* log base 2 of <exp> *)

Instructions

The following instruction names are recognized ('/' indicates alternative instruction names with identical meaning):

aaa	fdiv	fptan	jcxz	out
aad	fdvp	frndint	je	pop
aam	fdivr	frstor	jb	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	fdiv	fstenv	jne	repne
cld	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	fild	fxch	les	shr
dec	fildl	fxtract	lock	stc
div	fildcw	fyl2x	lodsb	std
f2xml	fildenv	fyl2xpl	lodsw	sti
fabs	fildl2e	hlt	loop	stosb
fadd	fildl2t	idiv	loope	stosw
faddp	fildlg2	imul	loopne	sub
fbld	fildln2	in	mov	test
fbstp	fildpi	inc	movsb	wait
fchs	fildz	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	

Additionally, a number of instructions have duplicate names with identical meanings. These are:

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
jle = jnge	repne = repnz
jle = jng	shl = sal
jnc = jae = jnb	xlat = xlatb

Symbol Table

The symbol table is discarded when a new section is started, except for the global equates which come before the module header. This means that symbols may be redefined. If a public symbol is redefined, the linker will detect this and will issue a warning.

If external objects are addressed, these should be defined in an **extrn** declaration. For example:

```
section; ...
extrn x$p
...
call far x$p
...
```

If objects are to be addressable outside the current section, the name should be preceded by a **public** declaration. For example:

```
public x$p: push bp
...
```

NOTE: An assembly procedure that will be called from a TopSpeed Modula-2 program should be declared as **public**.

Operand Sizes

In most cases, the operand size (byte, word, etc.) will be implied, and the size will be correctly determined by the assembler. However, if there is any chance of ambiguity, you must use a specifier to indicate the operand size. For example:

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

Jumps and Calls

Jumps and calls default to the smallest possible case. For instance, unless a label is defined before use, the assembler will assume it is Near and in the same segment.

If a jump goes out of range a specifier should be added, for example

```
jb near $loop
```

Strings

Single character string constants are treated as numbers

If a segment override is required for a string operation it has to be done using **db**, for example:

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

References

Forward references are assumed to be labels in the currently selected segment.

Equated symbols may be redefined; labels may not be redefined.

Non-8086 Instructions

Only 8086/8088/8087/80287 operations are supported. Opcodes for 286/386/387 may be simulated using **db**.

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

foption 0 (* causes floating point emulation; this does not affect **WAIT**, which is *never* emulated. Use **FWAIT** for emulated waits *)

foption 1 (* suppresses generation of emulation fixups *)

foption 2 (* suppresses generation of waits and fixups *)

The default is **foption 0** – that is, with floating point emulation.

Variables

Variables can be defined using the **db**, **dw**, or **org** directives:

db — define byte

dw — define word

org — reserve some (uninitialized) memory (Note: no \$+ is needed)

Example:

```
x: db 7      (* define a variable, x, with initial value of 7 *)
y: dw 0, 7   (* define a word, y, with initial byte values of 0, 7 *)
z: org 10    (* define a 10-byte, uninitialized variable named z *)
```

Miscellaneous Points

Statements such as

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

must be changed to

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

The following are regarded as separate instructions: **rep/rep/repne/lock**. They must be separated from other elements by a semicolon. For example:

```
rep; movsb (* note the semicolon *)
```


Naming Conventions

With the assembler, you can have almost complete control over the layout and organization of your program. It is, however, important to keep naming and calling conventions in mind if the assembler module is to be used with other modules.

The naming conventions for variables and procedures in object files for use with TopSpeed Modula-2 are as follows:

- Variable names are prefixed with the module name and an @-sign. For example, a variable named **Var1** in the definition module **Search** will have the name **Search@Var1**.
- The conventions for procedure names depend on whether the procedure is to be called with FAR or NEAR calls.
 - For FAR calls, the module name is followed by a \$ character. For example, the FAR procedure **FoundIt** in module **Search** would have the name **Search\$FoundIt** in this case.
 - For NEAR calls, the module name is followed by an @-sign. For example, procedure **Intr** in module **AsmLib** would be **AsnLib@Intr** in this case.
- Initialization code for each individual module is named using the module name, followed by '@' immediately after the name. For example, a module named **Search** would have initialization code named **Search@**.

TopSpeed Assembler Error Messages

The error messages for the TopSpeed Assembler are summarized in this section, along with their probable causes. Internal errors (which should not occur) are not discussed.

'Syntax error'

The program does not conform to the syntax. This could be for many reasons. Refer to the syntax description.

Example:

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

Correction:

```
db 0,0
```

'Unknown/misspelled instruction mnemonic'

The instruction is not known to the assembler.

Examples:

```
dd 0          (* dd not implemented in this assembler *)
movb          (* wrong spelling *)
```

Corrections:

```
dw 0,0
movsb
```

'Wrong number of operands'

There are too few or too many operands.

Examples:

```
mul ax,bx      (* ax is implicit *)
fadd           (* float instructions require explicit operands *)
rep movsb      (* rep is a separate instruction *)
ret            (* ret requires a parameter *)
```

Corrections:

```
mul bx
faddp st(1),st(0)
rep ; movsb
ret 0
```

'Incorrect operand(s)'

The operands for the instruction are not correct.

Example:

```
rol ax,5      (* 8086 can only do 1-bit constant rotates *)
```

Correction:

```
mov cl,5; rol ax,cl
```

'Missing byte/word specifier'

There is ambiguity in whether the operation is byte/word.

Examples:

```
mov [si],0
add [bp][6],15
fld st(0), es:[0]
```


Corrections:

```

mov byte [si],0
add word [bp][6],15
fld qword st(0), es:[0]

```

'Undeclared byte operand'

A byte operand cannot be a label, so must be declared before use.

Example:

```
mov al, x
```

Correction:

```

x = 99H
mov al, x

```

'Label not in expected segment'

Due to the one-pass high-speed design of the assembler, when a label is used before it is defined the assembler assumes that the segment in which it is defined will be the same as the first segment where it is mentioned.

Example:

```

segment code (CODE, 28H)
  mov ax, [x]
segment data (DATA, 28H)
x: dw 0

```

Correction:

```

segment data (DATA, 28H)
x: dw 0
segment code (CODE, 28H)
  mov ax, [x]

```

'Bad operand for seg operator'

The seg operator requires a label as operand.

'Bad operand for select'

Select requires a segment as the operand.

'No segment defined'

At least one segment must be defined before any instruction/data/label statements can occur.

'Bad operand for group'

Groups require segment operands.

'Not a segment register'

A segment override must be a segment register.

Example:

```
mov ax, dx:[bx]
```

Correction:

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

'Operand may not be label'

The operands for arithmetic operators, **org**, and **equates** require operands which are pure numbers. This can also happen because undefined symbols are assumed to be labels.

Example:

```
x = y + 5  
y = 6
```

Correction:

```
y = 6  
x = y + 5
```

'More than one label in address'

There should be only one label in an address.

Example:

```
x: dw 0  
y: dw 0  
mov ax, [x][y]
```

'Not an index register'

Only **bx**, **si**, **di** can be used as index registers.

Example:

```
mov ax, [dx]
```

Correction:

```
mov ax, [di]
```

'Unexpected end of file'

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

'Token too big'

(Most unlikely) Identifiers/strings are limited to some ridiculously large limit.

'Bad char in hex constant'

Hex characters must be in upper case.

Example:

```
x = 0aaH
```

Correction:

```
x = 0AAH
```

'Bad char in decimal constant'

Only digits are allowed in a decimal constant.

Examples:

```
x = 9A5
```

```
y = 99h (* upper case H required for hex constant *)
```

Correction:

```
x = 95
```

```
y = 99H
```


'Mismatched string constant'

A string constant must be terminated before end of line.

Example:

```
db 'hello , 0
```

Correction:

```
db 'hello' , 0
```

'Mismatched comment'

A comment was not terminated before the end of the file. Unfortunately this can be difficult to find. Use the editor to check all the comments.

'Bad character'

Some character was found which the assembler does not allow.

Example:

```
!x = 10
```

Correction:

```
_x = 10
```

'Scope table full / Name table full /

Too many identifiers'

(These are unlikely to occur!) Use **section** to divide the module up, or split into several modules.

'Store: Limit exceeded'

Also too many identifiers (Also unlikely to occur).

'Store: Out of memory(1/2)'

Not enough free memory. (Very unlikely) Try running assembler from command line instead.

Chapter 6

Interfacing Assembler to Other Languages

In the preceding chapter, the fundamentals of the TopSpeed Assembler were outlined. However, although useful on its own, code written and assembled with the TopSpeed Assembler can be easily interfaced to other languages – including, of course, JPI's Modula-2!

The process of calling procedures and accessing variables in different modules written in different language is very simple. Nevertheless, you should understand the previous chapter thoroughly, before continuing with this one.

There are basically just five steps to producing a Modula-2 program that uses an assembly language module, these are:

- Write the Modula-2 main program
- Write the Assembler program
- Write the definition file
- Assemble the Assembler program to a .OBJ form
- Compile and link the main program

Each of these steps should be familiar to you by now. (You can do the first three in whichever order you prefer.) Individually, they are simple, tying them all together sometimes causes concern!

This chapter will outline the main conventions used when interfacing to other languages. The chapter continues with a complete example program written in TopSpeed Modula-2 and assembler.

Calling Conventions

The calling conventions for procedures and function procedures are as follows:

- Any value parameters are pushed onto the stack, and the minimum size for a parameter is 2 bytes. Thus, 1-byte values will take up two bytes on the stack.
 - Open array value parameters are an exception to this because they are passed by reference (4 bytes), and are copied to a safe place by the called procedure. Open array parameters have their size pushed on the stack before the address (2 additional bytes, to give a total of 6 bytes).
- **VAR** parameters are passed by reference (4 bytes). Open array **VAR** parameters have their size pushed before the address — for a total of 6 bytes.
- The order in which parameters are pushed on the stack is the order in which they appear in the source code, from left to right.
- Function results are returned in various registers or on the stack in case of structured results.
 - Based pointers, **INTEGER**, **CARDINAL**, **SHORTCARD**, **SHORTINT**, **CHAR**, **BOOLEAN**, and enumerated values are returned in **AX/AL**.
 - Absolute pointers, **LONGINT** and **LONGCARD** are returned in **DX:AX**.
 - **REALs** are returned on top of the 8087 stack.
 - Any **ARRAYs** and **RECORDs** are returned on the 8086 stack. Space is reserved by the caller immediately, before any parameters are pushed.
- Procedures are not allowed to change **SI**, **DI**, **BP**, **DS**, **ES** (unless otherwise specified, although **BP** must always be saved). The preceding set of five registers are the default registers to be saved. (See the discussion of compiler options in Chapter 7 of the *TopSpeed Modula-2 User's Manual* for more information.)

A Short Example

Following from the earlier short example program which moves the letter 'J' into **AL**, and then returns from a far call. The procedure is called **main**, and is declared as public and far. This allows it to be called from another module. In this case, we want **main** to return a character, 'J', when called. Therefore, a definition module containing this information must be written.


```
DEFINITION MODULE example;
IMPORT AsmLib;
```

```
PROCEDURE main() : CHAR;
```

```
END example.
```

The assembler program is still:

```
module example
```

```
segment C_example(FCODE,28H); group G_example(C_example); select C_example;
```

```
LetterJ : db 'J'
```

```
public example$main:  mov ax,cs:[LetterJ]  (* public Far procedure,      *)
                      (* which can be called      *)
                      (* from another module      *)
                      ret far 0            (* this returns from the    *)
                                         (* far call in the M2 program *)
```

```
end
```

The JPI Modula-2 program which calls a procedure `main` and then prints the returned value 'J' is below:

```
MODULE ex;
```

```
FROM example IMPORT main;
FROM IO IMPORT WrStr;
```

```
VAR result : CHAR;
```

```
BEGIN
```

```
  result:=main();
  WrStr(result);
```

```
END ex.
```

When the Modula-2 program calls procedure `main`, it executes a far call. The assembly language program must have a suitable return instruction with a specifier (since the assembler does not know that the procedure is to be the subject of a far call). The far specifier directs the assembler to pop the return address from the stack. If `far` is not specified, `near` will be assumed and the program will crash.

Assembling the File

To create an `.OBJ` file, use the assembler, `M2ASM.EXE`, included in your TechKit. To do this, type

```
m2asm example.a
```

The assembler will create a file, `EXAMPLE.OBJ`, that you can then use in your Modula-2 programs.

It should not be necessary to point out that this program is completely useless, apart from providing a simple example of interfacing JPI's Modula-2 to an assembler module. Rest assured, a more complex example will follow shortly.

Initialization Code for Assembler Modules

If a module in assembler needs to initialize something before the procedure in the module can be called, this can be done by putting the code in a special segment called **INITCODE**. This segment has to be public and byte aligned; it must be in class **CODE** and must belong to a group called **G_CODE**.

The initialization code in **INITCODE** should have a public label (**<modulename>@**). If this module is using code in other modules, it should have an external declaration of that module, in the same section. This is necessary to ensure correct initialization.

Depending on whether or not the assembler code is to be used in a small code model, the code should be in a segment called **C_CODE**, which must be public, in class **CODE** and must belong to a group called **G_CODE**. Or the code must be in a segment called **C_<module name>**, which must be public, in class **FCODE** and which must belong to a group called **G_<module name>**. Data should be put in a segment called **D_<module name>**, which must be public and in class **M_DATA**.

To create such files, use the **M2ASM.EXE** assembler included with the TechKit. You can also use Microsoft **MASM** version 4.00 or later. Earlier versions of **MASM** do not handle groups correctly.

A More Complex Example

You can create an assembler file and make use of this file in your Modula-2 programs. First, create the assembler file. To create an **.OBJ** file, assemble the file using **M2ASM.EXE**. To make the file accessible as a module, you also need to create a **.DEF** file for the assembler file. This **.DEF** file will specify the public data structures and procedures from the assembler file.

The previous section's simple example will have provided you with the basics required to write an assembly module for a Modula-2 program. In this section, you'll find a more complex program, providing examples of all segment types, including **INITCODE**.

The following listing shows the definition part (let's call it **NAUGHTY.DEF**) for an assembler file. The module has one public procedure, **DosCount**, which returns a **CARDINAL** indicating the number of times the DOS function handler has been called since the **Naughty** module was installed.

The module also specifies two **LONGCARD** variables:

Tick[40H:108]

The number of times the system clock has ticked since midnight, with ticks occurring 18.2 times per second.

This value is associated with an absolute address — beginning at the location [40H:108], where 40H is the segment address, and 108 is the offset.

InstallationTime The number of ticks since midnight — that is, the value of **Tick** — at the time the **Naughty** module was installed.

```

DEFINITION MODULE Naughty;
IMPORT AsmLib;
(* Give linker information about the fact that Naughty uses AsmLib *)
(* This is only guaranteed to work if the main module is in MODULA-2 *)

PROCEDURE DosCount() : CARDINAL;
(* Returns the number of times interrupt 21H has been called since *)
(* this module was installed *)

VAR Tick[40H:108] : LONGCARD;
    (* Number of timer ticks since midnight; 18.2 ticks per second *)

    InstallationTime : LONGCARD;
    (* Timer tick at installation *)

END Naughty.
```

The following listing contains the assembler file, **Naughty.A**. The listing illustrates the format of a file for use with Modula-2 programs, and also provides an example of the TopSpeed assembly language “in action.”

```

module Naughty

segment C_NAUGHTY(FCODE,28H); group G_NAUGHTY(C_NAUGHTY); select C_NAUGHTY;

Count:   dw 0    (* Keep track of how often int21 has been called here *)
Old21:   dw 0,0  (* Store old interrupt 21H here *)
OldTerm: dw 0,0  (* Store old termination procedure here *)

public Naughty$DosCount:
    mov ax,cs:[Count];    (* load current counter contents into ax *)
    ret far 0             (* and return back to calling routines *)

Trap:    (* enter here on interrupt 21H *)
    inc word cs:[Count]    (* increment counter *)
    jmp far cs:[Old21]     (* pass interrupt onto old handler *)

Term:    (* remove interrupt 21H trap *)
    xor ax,ax              (* clear ax with an exclusive OR op. *)
    mov ds,ax              (* clear ds *)
    pushf                  (* save interrupt enable state *)
    cli                    (* disable interrupts while modifying *)
    mov ax,cs:[Old21]
    mov [21H*4],ax
    mov ax,cs:[Old21][2]
    mov [21H*4][2],ax
    popf                   (* restore previous interrupt enable state *)
    jmp far cs:[OldTerm]   (* chain to previous termination procedure *)

extrn AsmLib$Terminate
```



```
public Naughty$Install:
```

```
    (* Save old interrupt 21H address *)
```

```
    pushf
```

```
    cli
```

```
    xor ax,ax
```

```
    mov ds,ax
```

```
    mov ax,[21H*4]
```

```
    mov cs:[Old21],ax
```

```
    mov ax,[21H*4][2]
```

```
    mov cs:[Old21][2],ax
```

```
    (* Install new Interrupt 21H *)
```

```
    mov ax,Trap
```

```
    mov [21H*4],ax
```

```
    mov [21H*4][2],cs
```

```
    popf
```

```
    (* Install Termination Procedure Term *)
```

```
    push cs
```

```
    mov ax,Term
```

```
    push ax
```

```
    push cs
```

```
    mov ax,OldTerm
```

```
    push ax
```

```
    call far AsmLib$Terminate
```

```
    ret far 0
```

```
section; segment D_NAUGHTY(M_DATA,28H); select D_NAUGHTY;
```

```
public Naughty@InstallationTime: org 4
```

```
(* declare segment that contains any code that is to be executed first *)
```

```
section; segment INITCODE(CODE,28H); group G_CODE(INITCODE);
```

```
    segment D_NAUGHTY(M_DATA,28H); select INITCODE;
```

```
extrn Naughty$Install; extrn Naughty@InstallationTime
```

```
extrn asmlib@
```

```
public Naughty@:
```

```
    mov ax,D_NAUGHTY
```

```
    mov ds,ax
```

```
    mov ax,40H
```

```
    mov es,ax
```

```
    pushf
```

```
    cli
```

```
    mov ax,es:[108]
```

```
    mov [Naughty@InstallationTime],ax
```

```
    mov ax,es:[110]
```

```
    mov [Naughty@InstallationTime][2],ax
```

```
    popf
```

```
    call far Naughty$Install
```

```
end
```


Using Naughty

The following program shows how to use the **Naughty** module in a Modula-2 program. The program calls the DOS function handler (interrupt 21H) 1000 times — in order to check the system time. The program also shows how straightforward it is to import functions or variables from an assembler module.

```

MODULE Naut;

(* use the assembler module *)
FROM Naughty IMPORT DosCount, InstallationTime, Tick;
FROM IO IMPORT WrCard, WrLn, WrLngCard;
FROM Lib IMPORT Dos;
FROM SYSTEM IMPORT Registers, DI, EI;

VAR Count1, Count2, Looper : CARDINAL;
    reginfo : Registers;
    T : LONGCARD;

BEGIN
    reginfo.AH := 2CH;          (* load ah with function for time *)
    Count1 := DosCount();      (* get start time from assembler proc. *)

    FOR Looper := 1 TO 1000 DO (* call system time 1000 times *)
        Dos( reginfo);
    END;

    Count2 := DosCount();
    WrCard( Count1, 10);
    WrCard( Count2, 10);
    WrLn;

    DI;                        (* disable interrupts *)
    T := Tick;                 (* load number of ticks *)
    EI;                        (* enable interrupt mask again *)
    WrLngCard( T - InstallationTime, 6);

END Naut.

```

This program produces the following output, indicating that interrupt 21H was called 1000 times during the program, and that 7 ticks occurred between the time **Naut** was installed and the end of the program was reached.

```

      0      1000
7

```

Storage Layout for the Naut Program

The following listing shows the storage layout for the **Naut** program, which used the **Naughty** module. Notice that the **C_NAUGHTY** segment is in the same class as the other modules. Notice also there is a data segment associated with the module — **D_NAUGHTY** — but there are no **S_** and **K_** segments, since the module has neither segment constants nor structured constants.

Start	Stop	Length	Name	Class	Group
00000H	00083H	00083H	INITCODE	CODE	G_CODE
00083H	006E6H	00663H	C_CODE	CODE	G_CODE
006E6H	006F6H	00010H	ENTERCODE	CODE	(none)
006F6H	00700H	0000AH	S_CODE	CODE	G_CODE
00700H	00772H	00072H	C_Naut	FCODE	G_Naut
00772H	00778H	00006H	S_Naut	FCODE	G_Naut
00778H	00778H	00000H	K_Naut	FCODE	G_Naut
00778H	00788H	00010H	C_Lib	FCODE	G_Lib
00788H	0078CH	00004H	S_Lib	FCODE	G_Lib
0078CH	0078CH	00000H	K_Lib	FCODE	G_Lib
0078CH	00ADAH	0034EH	C_IO	FCODE	G_IO
00ADAH	00ADEH	00004H	S_IO	FCODE	G_IO
00ADEH	00B00H	00022H	K_IO	FCODE	G_IO
00B00H	00B62H	00062H	C_NAUGHTY	FCODE	G_NAUGHTY
00B62H	00C7CH	0011AH	C_Str	FCODE	G_Str
00C7CH	00C7EH	00002H	S_Str	FCODE	G_Str
00C7EH	00CA8H	0002AH	K_Str	FCODE	G_Str
00CA8H	00CA8H	00000H	DATA	DATA	DGROUP
00CB0H	00CB0H	00000H	DataStart	Dummy	DGROUP
00CB0H	00CB0H	00000H	BSS	BSS	DGROUP
00CB0H	00CCEH	0001EH	D_Naut	M_DATA	(none)
00CCEH	00CD2H	00004H	D_Lib	M_DATA	(none)
00CD2H	00E0AH	00138H	D_IO	M_DATA	(none)
00E0AH	00E12H	00008H	D_DATA	M_DATA	(none)
00E12H	00E16H	00004H	D_NAUGHTY	M_DATA	(none)
00E16H	00E18H	00002H	D_Str	M_DATA	(none)
00E20H	04E20H	04000H	STACK	STACK	(none)
04E20H	04E20H	00000H	HEAP	HEAP	(none)

Chapter 7

Creating TSR Programs

The TechKit contains all the definition and implementation files required for a TSR module. This module enables you to create resident programs, similar to SideKick and ProKey, that are loaded into memory. When first run, these programs terminate but remain in memory — as an extension to the part of DOS that remains resident. Such a program remains in an inactive state until invoked through a special 'hot-key' sequence.

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

- Redefine the interrupt services table so that the keyboard interrupt entry points to a short routine. This routine compares keystrokes to the 'hot-key,' passing the keystrokes on to DOS if no match occurs. If the keystrokes match the 'hot-key,' the routine activates the main program.
- Establish the size of the routine to remain resident, along with any stack and heap required. This figure is rounded up to the nearest paragraph, and is used when calling DOS to terminate and keep the program.
- Call DOS (function 31H) requesting that the current program be terminated, thereby freeing all memory other than the required resident portion.

The code for a TSR program remains resident in a portion of the program area in working memory. The program is suspended until a particular key, the **HotKey**, is pressed. At that 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 will look similar to the following:

Interrupt Services Table
Resident hidden DOS programs (IBMI0.COM & IBMDOS.COM)
COMMAND.COM
Resident part of your TSR, size allocated with termination call
Rest of memory

To see how the TSR module works, the source code is provided in **TSR.MOD**. However, be careful if you intend to modify the source code.

Activating a TSR Program

The HotKey to activate the TSR program will generally be a combination of one or more "shifted" keys — **(Ctrl)**, **(Alt)**, **(RShift)**, and **(LShift)** — When one of these keys is pressed, a bit is set to ON in a **status mask** at a particular address. In addition, the program may require another key to be pressed — this will return a **scan value** associated with the key. This combination of status mask and scan values constitutes the HotKey which will activate the TSR program.

In the 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.

When writing a TSR program, you will need to specify the mask and scan values that will make up your HotKey. These values will be 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 will do this for you. After **Install** has been called and has done its work, the program is in memory — suspended but ready to be activated.

The calling convention for **Install** is:

```
PROCEDURE Install ( P      : PROC;      (* Procedure to be invoked      *)
                   KBF     : KBFlagSet; (* Shift state to invoke      *)
                   Scan    : SHORTCARD; (* Scan code to invoke (0=All) *)
                   heapsize: CARDINAL; (* Minimum heap required      *)
                   ) ;
```

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*).

The following listing shows how to call **Install**.

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

This call specifies the following:

- The TSR program will be activated by pressing **Ctrl LShift A**
- The program will need 4K of heap space (16 * 256)
- When activated by the HotKey combination, the program calls the global procedure **TheAction** to perform the program's task

Procedure **Install** will not return until the TSR is deinstalled. That is, the program is unloaded from memory. **Install** remains active even after **TheAction** has finished any processing activated by the HotKey.

TABLE 7.1.

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

A Short 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.

```

(* does the work *)
 (* Ctrl, Left Shift *)
 (* A *)
 (* use 2K of heap *)

Cautions about TSR Programs

Be careful when writing TSR programs. Keep in mind 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 then your TSR pops up and tries to use the same MS-DOS variables, things will get interesting, and the system will probably crash. The TSR module will only pop up your program when it is safe to use DOS calls above 0CH.

There are several commands and functions that you cannot use from a TSR. The following list summarizes these restrictions and cautions.

- You cannot use DOS functions below 0DH. This includes routines **IO.RdKey** and **IO.KeyPressed**
- You should not use the Scheduler provided in the **Process** module.
- You should ensure that sufficient heap and stack space has been allocated (*see below for how to do this*).
- Switch run-time checking off. If a run-time error does occur, then the machine will require a hard reset.
- If you plan on using more than one TSR program resident at any one time, they may clash and require loading in a particular order. This will probably need some experimentation.
- When a TSR program starts working, it will not check to see what is displayed on the screen before starting to write. This may, unless you take precautions to prevent it, corrupt the screen. To avoid corrupting what is displayed on the screen, 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, make sure it is well-behaved towards system calls, interrupts and resources — in order to minimize the likelihood of clashing with any other programs.

TSRCALC.MOD provides a more complete example of a TSR program. That program uses a window for doing your calculations. Notice that the window is defined in the

main **TSRCALC** program, *not* in the procedure **RunCalc** (that executes when the TSR program is activated.) **RunCalc** simply uses the window for its screen I/O.

When writing TSR programs, you will need to take careful control of the size of the program's heap and stack. The size of the heap before the **TSR.Install** procedure is called is not relevant. Only the amount of heap requested through **TSR.Install** will be available to the program.

If the heap size passed to **TSR.Install** is too small for the objects that are already allocated on the heap, the program will not install, and will generate a

not enough storage

error message. You can set the stack size by using the **\$S** compiler directive in the program's main module.

You should never run programs that will terminate and stay resident from within the TopSpeed environment. If you do, the program will become resident within TopSpeed's memory space, typically causing the machine to lock up. The **TSR.Install** procedure terminates (without installing), and displays an error message if you try to install from within the environment.

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 brief example above deactivates itself immediately after writing "Hello" on the screen. You need not do anything for this to happen. Other programs — such as the **TSRCALC** included with the TechKit — continue running until you indicate that you are done. Generally, pressing **Esc** will accomplish 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 up the memory in which the program had been residing.

To terminate the TSR program, you must call **DeInstall** or **HALT**. Such a call will restore all interrupt traps, and will free the memory that had been allocated to

the 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 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.

Notice that the simple TSR program used earlier does not call **DeInstall**. This means the program remains resident until you reboot. Other programs will allow you to terminate them. For example, pressing **Alt X** while **TSRCALC** is active will terminate the program completely. Pressing **Alt X** produces a call to **DeInstall**.

If your TSR program cannot guarantee (to itself) that it will be the last program in memory, then it may be wise to prevent the user from being able to terminate the program.

The TSR Module

The **TSR** module is interesting for several reasons. For one thing, it represents a good example of the advantages of being able to keep procedures and data hidden within an implementation module.

The definition part of the module is quite sparse. There are only two procedures defined — **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 the mess 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've compiled the calculator, you can install it by typing

```
tsrcalc
```


Using the RS-232 Port

The **rs** module provides the procedures and data structures you need to use the RS-232 serial communications port on your PC. The module definition contains several procedures for using and monitoring this port. You can use communications port 1 or 2.

Serial communication generally takes place asynchronously. That is, one event will dictate the timing of another. This is in contrast to synchronous communications (usually used for communication with 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 while waiting for an asynchronous event to occur, communications programs are usually 'interrupt-driven.' That is, the programs will act on an interrupt from the port. The serial port hardware will generate 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 is called **polling**, and is really suitable only for slow data rates. For this reason, polling is not used in the **rs** module.

Thus, interrupts play an important role in the communication process. Interrupts are particularly important when multiple processes may be running at the same time. In that 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 have been 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 will signal the status of the serial port. That is, they will signal whether data can be sent, have arrived, or if a buffer is empty or full.

The **rs** module is designed to handle communication ports 1 and 2 on the PC. The module source can easily be modified to provide support for ports 3 to 8 if necessary. To correctly drive **COM1** or **COM2**, you need to know the interrupt they use and the port address they occupy. These are:

Port	Interrupt	Address
COM1	4	3F8H
COM2	3	2F8H

It should also be mentioned that the PC'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 communications (1200 baud and over). When you need speed, use the **rs** module!

The rs.Def File

The definition part of the **rs** module defines two data types:

- pt** the parity being used
- w1** the number of data bits being used in the transmission

Initializing the Port

Two of the **rs** procedures are used to set up the environment to allow proper use of the port.

Install

```
PROCEDURE Install( Port : CARDINAL);
```

This procedure installs the appropriate interrupt handlers for the specified port. It must be called before any of the other **rs** procedures can be called.

Example:

The following program statement will install the necessary interrupt handlers to enable a user to work with the second communications port (COM2):

```
Install(2);
```

Install2

```
PROCEDURE Install2( Port,Intr : CARDINAL);
```

This is a secondary installation routine in which the interrupt to be handled must be specified (using the **Intr** argument). This is mainly to allow the access to communication ports other than COM1 and COM2 (or any other non-standard serial port).

On several machines COM3 and COM4 can be used. To install COM3:

```
CONST Port3 = 404H;  
Install2(Port3,4);
```

And, to install COM4:

```
CONST Port4 = 406H;  
Install2(Port4,3);
```

Init

```
PROCEDURE Init( Baud      : CARDINAL;  
                WordLength : wl;  
                Parity    : pt;  
                OneStopBit : BOOLEAN;  
                HandShake  : BOOLEAN);
```

The parameters have the following ranges:

Baud	the baud rate, which can be up to 115,200 on fast machines.
WordLength	the word length to be used, which can be between 5 and 8 bits.
Parity	the parity, which can be any of: None , Even , Odd , Mark or Space .
OneStopBit	a Boolean switch, which can specify either one stop bit or none.
HandShake	a Boolean switch, which can set CTS handshaking to ON or OFF.

This procedure initializes the port specified in **Install** and sets-up the various port parameters to be used according to the specified settings passed as arguments. **Baud** is the baud rate (usually in the range between 300 or 1200 baud). **WordLength**

specifies the number of data bits in each packet sent; **Parity** specifies what type of parity checking (if any) will be used.

The two **BOOLEANS** specify whether each packet of information sent should be terminated by a single stop bit, and whether the two computers will use **handshaking** to coordinate their communication. (The handshaking used is hardware-based, using the CTS RS-232 line.)

During initialization, these settings are sent to the port being used. The port being used has already been specified in a call to **Install**.

Example:

```
Init( 1200, 7, Even, TRUE, TRUE);
```

initializes communication at 1200 baud, using seven data bits and one stop bit, with even parity, and using handshaking.

Keeping Track of Buffers

Buffers are set up to handle the received and transmitted data. Initially, these buffers are set to 256 bytes long. This size can be increased by modifying the **BufferSize** parameter in the **rs** module (line 25).

The module includes three procedures that enable you to determine information about the buffers used in receiving or sending information.

RxCount

```
PROCEDURE RxCount () : CARDINAL;
```

This procedure returns the number of bytes in the receiving buffer.

Example:

```
VAR NrIn : CARDINAL;
...
NrIn := RxCount();
```

TxCount

```
PROCEDURE TxCount () : CARDINAL;
```

This procedure returns the number of bytes still to be sent — that is, remaining in the transmitting buffer.

Example:

```
VAR NrLeft : CARDINAL;
...
NrLeft := TxCount();
```

TxFree

```
PROCEDURE TxFree() : CARDINAL;
```

This procedure returns the number of bytes free in the transmitting buffer. This buffer is particularly important in a multitasking program which uses the RS-232 interface to transmit packages of information — so the user can ensure that a package is not split up during the transmission process.

Example:

```
VAR NrFree : CARDINAL;
...
NrFree := TxFree();
```

Sending and Receiving

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.

Send

```
PROCEDURE Send ( Buf : ARRAY OF BYTE; Len : CARDINAL );
```

This procedure sends **Len** bytes to the port being used. The information being sent is in **Buf**.

Example:

```
VAR XBuff : ARRAY [ 300] OF BYTE;
...
Send ( XBuff, 200);
```

sends 200 bytes, taken from the array **XBuff**. These bytes are sent to whatever port (1 or 2) is being used.

Receive

```
PROCEDURE Receive ( VAR Buf : ARRAY OF BYTE; Len : CARDINAL );
```

This procedure receives **Len** bytes through the port being used. The information being received is stored in **Buf**.

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 two procedures let you send a break signal from one computer to the other, and let you test whether a break signal has been received by your computer.

Break

```
PROCEDURE Break ( Time : CARDINAL );
```

This procedure sends a break signal lasting **Time** *milliseconds*.

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 such as:

```
While TXCount() # 0 DO
  (* wait until transmit buffer is empty *)
END;
Break (500);
```


BreakTest

PROCEDURE BreakTest():BOOLEAN;

This procedure returns **TRUE** if a break signal has been received, and **FALSE** otherwise.

An Example Program

The following program is a very short program that 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 and hangs-up the telephone (using the Hayes compatible ATH0 command).

```

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                    (* if the i/p queue is empty, *)
      rs.Receive(inp, 1);                    (* then receive one character *)
      IO.WrChar(inp);                        (* and write it to screen *)
    END;
  END;
END Example.
```

The RDemo Program

The RDemo program included in your TechKit is a simple driver for the RS-232 port. The program uses the **rs** module.

RDemo 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 will transmit 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 might want to make the program more sophisticated and robust.

You can adapt the program easily. For example, simply change the **PhoneNr** string to call whatever number you want. Thus, you could log onto BIX, for example, simply by changing the number and recompiling.

Miscellaneous Modules

This chapter contains information about several other modules included in your TechKit.

BiosIO Module

The **BiosIO** module provides several low-level keyboard handling routines. The module includes four procedures and some data structures, including **KBFlagSet**, a set for specifying the value of the key status mask.

KeyPressed

```
PROCEDURE KeyPressed() : BOOLEAN ;
```

KeyPressed returns **TRUE** if a key has been pressed.

Example:

```
VAR HaveKey : BOOLEAN;  
...  
HaveKey := KeyPressed();
```


RdKey

```
PROCEDURE RdKey() : CHAR ;
```

This procedure reads a single key, but does *not* echo the character.

Example:

```
VAR X : CHAR;
...
X := RdKey();
```

If you type 'g,' the character will be returned as a result of the function call, and can be assigned internally to an appropriate variable. Your letter will *not* appear on the screen, however.

RdChar

```
PROCEDURE RdChar() : CHAR ;
```

This procedure reads a single key, and *does* echo the character.

Example:

```
VAR X : CHAR;
...
X := RdChar();
```

If you type 'g,' the character will be returned as a result of the function call. In this case, your letter *will* appear on the screen, however.

KBFlags

```
PROCEDURE KBFlags () : KFlagSet ;
```

This procedure returns information about the "shift" keys currently being pressed.

Example:

```
VAR MyFlags : KFlagSet;
...
MyFlags := KBFlags();
```


A Critical Error Handler

DOS interrupt 24H is the critical error handler that is called when a hardware error (such as an open disk drive door, a printer that is off-line, and so forth) occurs. The DOS version of this interrupt displays the

Abort, Retry, Ignore?

message when called.

Once the DOS interrupt handler has responded, your program is at the mercy of this handler. Chances are, your program will have to be terminated prematurely. Since there is no way to get back to your program before this happens, any open files will be lost.

You can avoid this by taking control of the interrupt vector associated with interrupt 24H. This way, your program can decide what actions are to be taken in the event of an error.

For example, you may want to give the user an opportunity to put the printer on-line, and then proceed — rather than terminating prematurely because the printer was not yet turned on.

Using your own custom-written critical-error handler has many benefits (not least of which is allowing you the luxury of being able to forget to put your printer on-line!). However, because DOS is called when one of these interrupts occurs, you cannot (since DOS is not re-entrant) use all the normal DOS calls. This means that Critical-Interrupt handlers are restricted to using only the following calls, all of which are accessed through the function dispatcher (Int 21H).

- Functions 01 through to 0CH. These are the character I/O calls to the keyboard, display, printer, and serial port.
- Function 30H, which gets the DOS version number. This function is available only with MS-DOS v2.0 or later.
- Function 59H, which Gets extended error information, and which is available only with MS-DOS v3.0 or later.

One last hitch to a beautifully written critical-error handler is that your handler does not remain resident. Rather, the handler be active only for the duration of the program which installed it. This is due to the way in which **COMMAND.COM** works: it restores Int 24H to the MS-DOS default whenever **COMMAND.COM** is in control. This means each application will require its own handler.

The **Int24** module supplied provides a simple example of how to take control of interrupt 24H.

The **Int24** module is very interesting in that there is nothing in the definition file! The initialization code in **INT24.MOD** takes control of the interrupt vector. After that, nothing else needs to be done: anytime interrupt 24H is activated, your handler will respond. To change the way in which the handler responds, you need to change the body of procedure **Int24Handler** in **INT24.MOD**.

Procedure **ForceSave** makes the compiler save all registers and returns the base of the saved registers. Notice that **ForceSave** is declared as a **FORWARD** procedure — in order to fool the compiler into accepting the way information is being transferred. If the procedure is not declared as **FORWARD**, only the registers that are actually used will be saved.

RegP1^ contains the calling registers. In particular, **DI** contains the code for the error.

Procedure **PopStack** can be used to return to the caller of the failed procedure, returning 255 in **AL** to indicate that the function is aborted. Otherwise **HALT** can be used to abort the entire program.

A Run-Time Error Handler

The **RTerror** module allows any run-time errors to be trapped and dealt with by a user's procedure, which is called when a run-time error occurs. The specified procedure will be called when any of the following errors occurs:

- Index out of range
- Arithmetic overflow
- Subrange value out of range
- Enumeration value out of range
- Dereference of **NIL** pointer

Two other run-time errors may occur which, because of their nature, cannot be handled by a user's routine. These are:

- Stack Overflow
- No **RETURN** From Function

When a run-time error occurs, the user routine will be called and supplied with the address at which the error occurred (both as an absolute memory location, and as a program-relative location), and the type of error which occurred. The routine may return **TRUE** if execution is to continue (ignoring the error), or **FALSE** if the run-time system's run-time error handling routines should be called. The routine may also **HALT**, or possibly **LongJmp** or **TRANSFER** to another location. If the

program returns FALSE to pass control to the Run-Time System Handler, all of the user-installed handlers are deinstalled.

Run-time errors are generated by an INT 4 instruction, which is followed by a one-byte code which designates the error which occurred. (These codes are selected to be equal to single-byte machine code instructions, so that disassemblies of the program do not show illegal instructions.) When the interrupt handler is entered, the return address on the stack is the address of the error code. To ignore the error, the interrupt routine simply adds one to the return address on the stack (so that the error code isn't 'executed'), and issues an IRET. The run-time error interrupts are positioned so that skipping past them causes execution to continue as though the error had not occurred.

The **RTerror** module exports the enumerated type **RTtype**, whose individual values represent the run-time errors which may be trapped. It also exports the **RTproc** type, which is the definition for the user-installed error handling procedure.

The **RTtype** is an enumerated type with form:

```
RTtype = (IndexOutOfRange,      (* past end of array *)
          ArithmeticOverflow,    (* overflow on arithmetic operation *)
          SubOutOfRange,         (* past end of subrange *)
          EnumValue,             (* illegal value for enumerated type *)
          NILusage);             (* de-reference of NIL pointer *)
```

The **RTproc** type is of the form:

```
RTproc = PROCEDURE (CARDINAL, CARDINAL, CARDINAL, RTtype) : BOOLEAN
```

in which the four parameters are:

1. the absolute segment where the error occurred;
2. the relative segment (0FFFFH if the error was outside of a normal program, which is possible with overlays);
3. the offset of the error;
4. the type of error;

It also has a return value which — if TRUE — causes program execution to continue; if the value is FALSE, then the normal run-time system error handler is called, and all user-installed handlers are deinstalled.

The two procedures that concern the user are **InstallRT** and **UninstallRT**. **InstallRT** is called with one argument — the name of the procedure that will handle the errors. If this procedure is called and there are already eight handlers installed, a NOP instruction will be performed. **UninstallRT** takes no parameters, and simply removes the handler currently in operation. If there are multiple handlers installed, the next one in the list will become effective. Trying to uninstall when there are no handlers installed will result in a NOP.

These two procedures are of the form:

```
PROCEDURE InstallIRT(handler : RTproc)
```

and

```
PROCEDURE UninstallIRT
```

Using Lotus/Intel/Microsoft Expanded Memory

The **LIM** module provides procedures for accessing and using expanded memory based on the Lotus/Intel/Microsoft standard. If your system does not have an expanded memory driver, the routines should not be called. The variable **LIMPresent** can be inspected to determine whether expanded memory is present.

The functions in this module use interrupt 67H, which gives you access to the LIM expanded memory. The calls are handled by the **Expanded Memory Manager (EMM)**, which serves as an intermediary between programs and the expanded memory.

Before calling interrupt 67H, a code is stored in register AH. The value stored in this register determines which EMS function is being called. For example, storing 40H in this register calls EMS function 1, which returns information about the status of the expanded memory. Most results from EMS functions are returned in the AX (mainly AH), BX, and DX registers.

Expanded memory is allocated to programs or processes by associating a handle with an area of expanded memory. This handle is returned to the process — just as a handle is returned when a file is opened. This handle should be used to refer to the expanded memory allocated for the process. Such “logical” storage is mapped to physical storage if material needs to be transferred to regular memory.

The initialization code first checks whether any expanded memory is present. If not, then none of the other procedures can be used.

GetStatus

```
PROCEDURE GetStatus() : SHORTCARD;
```

This procedure determines whether the expanded memory *hardware* is present and working.

Example:

```
VAR Stat : SHORTCARD;  
...  
Stat := GetStatus();
```

If this call returns 81H, then the expanded memory hardware is not working properly; if it returns 0, then the hardware is working.

GetPageFrame

```
PROCEDURE GetPageFrame() : CARDINAL;
```

A **page frame** represents physical memory associated with the EMM. When information needs to be transferred from the storage associated with an expanded memory handle to regular memory, the material is first mapped to the page frame. A page frame for expanded memory is divided into four 16K pages. These pages are used to map between the logical pages in expanded memory and the physical address spaces in the regular memory — that is, memory accessible to the processor.

The **GetPageFrame** procedure gets the segment address of the page frame being used in expanded memory. The segment information is returned by the procedure.

Example:

```
VAR PageSeg : CARDINAL;  
...  
PageSeg := GetPageFrame();
```

FreePages

```
PROCEDURE FreePages() : CARDINAL;
```

This procedure returns the number of available pages in expanded memory — that is, the number of pages that are still not allocated.

Example:

```
VAR NrFree : CARDINAL;  
...  
NrFree := FreePages();
```


AllocatePages

```
PROCEDURE AllocatePages( n : CARDINAL ) : CARDINAL;
```

This procedure allocates the specified number of logical pages in expanded memory, and returns a handle for the allocated memory.

Example:

```
VAR LIMHandle : CARDINAL;
...
LIMHandle := AllocatePages( 3 );
```

gets a handle (returns to **LIMHandle**) and allocates three expanded memory pages for this handle.

MapPage

```
PROCEDURE MapPage( PPage, LPage, Handle : CARDINAL ) : CARDINAL;
```

This procedure maps a logical page in expanded memory into one of the four physical pages in the page frame. **PPage** specifies the (physical) page number in the page frame, and must be between 0 and 3, inclusive. **LPage** specifies the logical page number in the memory associated with the **Handle**. **Handle** is the value returned by **AllocatePages**. **LPage** must be between 0 and N-1, inclusive, where N is the number of logical pages allocated with **AllocatePages**.

The value returned by **MapPage** is an offset, within the page frame, to the actual page. Thus, the page can be accessed as an address, using the following format and calls:

```
[GetPageFrame() : MapPage(...)]
```

(See the discussion of the **pointer constructor** in the *TopSpeed Modula-2 User's Manual* for more information about this construct.) Once this mapping has been carried out, you can access the page frame by using the segment address returned by **GetPageFrame()**.

Example:

```
VAR PageOfs : CARDINAL;
...
PageOfs := MapPage( 3, 5, LIMHandle );
```

This statement says to map logical page 5 to physical page 3 (which has offset 0C000H) in the page frame. The logical page comes from the storage associated with **LIMHandle**.

DeallocatePages

```
PROCEDURE DeAllocatePages( Handle : CARDINAL );
```

This procedure releases the logical pages allocated for **Handle**, and then releases **Handle**.

Example:

```
DeallocatePages( LIMHandle);
```

releases the pages associated with **LIMHandle**.

The TechKit also contains executable versions of some useful programs.

Analyze

The **ANALYZE**.EXE program determines and displays the logical dependencies among the modules in a program. It also prints out a list of circular imports, and a sorted list of circular dependencies. Thus, **ANALYZE** can also be very useful for diagnosing the results for diagnostic errors reported by the linker. The program gets its use information from the .OBJ file for the main module. The .OBJ file for the modules must be available to **ANALYZE**.

To use the program, type **ANALYZE**, followed by the name of the main module in the program you want to analyze. For example, to analyze the interdependencies among the modules used in the **NAUT** program in Chapter 3, type the following:

```
analyze naut
```

You can redirect output to a file using the DOS **>** or **>>** command. For example, to redirect the output to a file, **NAUT.ANL**, type

```
analyze naut > naut.anl
```

NAUT.ANL should be accessible in the current directory if redirection is required.

Chapter 10

Programs Included With the TechKit

The TechKit also contains executable versions of some useful programs.

Analyze

The **ANALYZE.EXE** program determines and displays the import dependencies among the modules in a program. It also prints out a list of circular imports, and a sorted list of time/date anomalies. Thus, **ANALYZE** can also be very useful for pinpointing the reasons for date/time errors reported by the linker. The program gets its first information from the **.OBJ** file for the main module. The **.OBJ** file for the module must be accessible to **ANALYZE**.

To use the program, type **ANALYZE**, followed by the name of the main module in the program you want to analyze. For example, to analyze the interdependencies among the modules used in the **NAUT** program in Chapter 3, type the following:

```
analyze naut
```

You can redirect output to a file using the DOS **>** or **>>** commands. For example, to redirect the output to a file, **NAUT.ANL**, type

```
analyse naut > naut.anl
```

M2.RED should be accessible in the current directory if redirection is required.

If **NAUT** is analyzed, the following output results.

Import analysis of naut

```
-----
naut      IMPORTS Lib, IO, SYSTEM, AsmLib, Naughty, Naut ;
Lib       IMPORTS SYSTEM, AsmLib, Lib ;
IO        IMPORTS Lib, SYSTEM, AsmLib, Str, IO ;
SYSTEM
AsmLib
Naughty
Naut      IMPORTS Lib, IO, SYSTEM, AsmLib, Naughty, Naut ;
Str       IMPORTS MATHLIB, Lib, SYSTEM, AsmLib, Str ;
MATHLIB
Date/Time Anomaly Check
-----
No date/time anomalies detected
Circular Import Check
-----
No circular imports detected
```

The **import analysis** lists implicit or explicit imports within each module, thus providing a dependency list. For example, the first line in the import analysis indicates that module **Naut** is dependent on modules **Lib**, **IO**, **SYSTEM**, **AsmLib**, **Naughty**, and on **Naut** itself.

No dependencies are listed for assembler modules or for non-TopSpeed Modula-2 modules. If module **A** is dependant on module **B** then the initialization part of **B** will be executed before the initialization part of **A** — unless there is a circular dependency (see below).

The **date/time anomaly** list shows date/time conflicts in files. The most recent files are displayed first, as these are the most likely to be in error.

A **circular dependancy** list results if a module is (directly or indirectly) dependent on a module that is dependent on the first module. If a module is part of a dependency “circle” then the module’s initialization order is undefined relative to the other members of that “circle.”

The format of a line in such a list is

```
CIRCLE: modulename1 > modulename2 { > modulenameN } > modulename1
```

ObjCompr

This program makes your object file somewhat smaller. The resulting files are still usable to TopSpeed Modula-2.

To use **ObjCompr**, enter the program name, followed by the name of the **.OBJ** file you want to compress. For example, the following command results in about a 9% decrease in the storage required for **Naut.OBJ**.


```
objcompr naut.obj
```

The **.OBJ** extension is necessary when using **ObjCompr**.

SetHeapSize

This procedure lets you specify the minimum and maximum heapsize of a Modula-2 program. The program takes **<MinAlloc>** and **<MaxAlloc>** as hex numbers representing paragraphs.

For example, to specify that the heap size for a program — let's say **NAUT.EXE** — should be between 400H and 800H (1K and 2K, respectively) paragraphs, type

```
setheaps naut.exe 400 800
```

Afterwards, the program will not load unless 400H paragraphs of heap will be available, and will never provide more than 800H paragraphs of heap.

Notice that the file name must end in **.EXE**. Notice also that the numerical parameters do not have an 'H' after the values. This is because the program only understands hexadecimal values, so there is no particular reason to use it here.

If you simply type the file name, with no numerical arguments, the program will allocate any pair of values between 0 and FFFFH paragraphs.

If you try to allocate too much storage the program will refuse to do it, and responds with a "Bad parameter" message.

Conclusion

That's it! As you have seen, TopSpeed Modula-2 can be a very powerful and versatile tool. The modules and programs included in the TechKit enable you to do just about anything you want, provided you are innovative enough.

With the introduction in this manual, and with your own experience using interrupts and other aspects of DOS, you should be able to explore the TechKit components and also the TopSpeed Modula-2 capabilities on your own.

Happy explorations.

Index

\$D Compiler Directive, 21, 22
\$G Compiler Directive, 7
\$M Compiler Directive, 9, 13,
21, 22
\$N Compiler Directive, 12

/J Compiler Option, 22

8086 Operands
Table, 34, 37
8087 Emulator, 17
8087 Opcodes, 43
8087 Registers
Table, 42
8087 Support, 16

Absolute Addresses
Creating Code with, 15
AllocatePages, 96
ANALYZE.EXE, 99
AsmLib Module, 16, 47
Assembler Modules
Initialization Code, 68

Assembler
Calling, 48
Calling Conventions, 66
Error Messages, 58
Instructions, 54, 56
Keywords, 51
Miscellaneous, 57
Naming Conventions, 7, 58
Strings, 56
Symbol Table, 55
Syntax, 51

Variables, 57
Assembling, 67
Auxiliary Flag, 35

Batch Linker, 22
BioSIO Module, 89
Break, 86
BreakTest, 87

Calls, 56
Far, 12
Near, 12
Carry Flag, 35
Class, 4, 6
Code and Data Segmentation, 9
Communications
Example, 87
Compiler Directive, 13
\$D, 21, 22
\$G, 7
\$M, 9, 21, 22
\$N, 12
Compiler Option
/J, 22

Suppress Libraries, 22
Critical Error Handler, 91

DeallocatePages, 97
Debugging Overlays, 29
Default Storage Layout, 4
Descriptor Tables, 34
Direction Flag, 34

Error Handler
Critical, 91

Run-Time, 92
 Error Messages
 Assembler, 58
 Expanded Memory
 LIM, 94

Flag

 Auxiliary, 35
 Carry, 35
 Direction, 34
 Interrupt Enable, 34
 Overflow, 34
 Parity, 35
 Sign, 34
 Trap, 34
 Zero, 35

Floating Point (8087)
 Instructions, 42

FreePages, 95

GetPageFrame, 95

GetStatus, 94

Group, 4, 6

HotKey, 73

IMPORT dependencies, 4

Init, 83

INITCODE, 4

Initialization Code

 Assembler Modules, 68

Install, 82

Install2, 83

Instruction Sets

 8088/8087, 33

Instructions

 8086, 36

 Floating Point (8087), 42

Int24 Module, 91

Interrupt Enable Flag, 34

IO.KeyPressed, 77

IO.RdKey, 77

Jumps, 56

KBFlags, 90

KeyPressed, 89

Keywords

 Assembler, 51

LIM, 94

Linking Large Programs, 21

M2ASM.EXE, 47

M2LOCATE.EXE, 15, 16

Make utility, 3

MapPage, 96

.MEM File, 15

Memory Addressing, 35

Module

AsmLib, 16, 47

BiosIO, 89

Int24, 91

Overlay, 20

OvlMan, 28

Process, 77

rs, 81

RTerror, 92

rtl, 47

SYSTEM, 16, 47

TSR, 73, 79

.OBJ File, 47

ObjCompr Program, 100

Opcodes

 8086, 38

 8087, 43

Operand Sizes, 55

Operands

 8088/8087, 36

Operator Precedence, 51

Order.A, 15, 47

Overflow Flag, 34

Overlay Module, 20

Overlays

 Debugging, 29

 Example, 23

 Implicit Assignment Rules,

 26

 Introduction, 19

- Miscellaneous, 29
- Restrictions, 25
- Small Segments in, 21
- OVL** Program, 20
- OvlDirectory.DEF**, 31
- OVLEXE** Program, 23, 26
 - Options, 27
- OvlMan** Module, 28
- OVLMAP**, 28
- OVLXREF**, 28
- OVn** Files, 21
- Parity Flag, 35
- Polling, 81
- Process** Module, 77
- Program Segments, 49
- PROM**
 - Putting Things in, 16
- RdChar**, 90
- RDemo** Program, 87
- RdKey**, 90
- Receive**, 86
- Registers, 33
- rs** Module, 81
- rs.DEF** File, 82
- RTerror** Module, 92
- rtl** Module, 47
- Run-Time Error Handler, 92
- Run-Time Library Calls, 10
- RxCOUNT**, 84
- Scan Codes
 - IBM PC, 76
- Segment Registers, 35
- Segment, 4
 - Code, 50
 - Data, 50
 - Program, 5, 49
- Selectors, 34
- Send**, 85
- SetHeapSize**, 101
- Sign Flag, 34
- Storage Layout
 - Default, 4
 - Example, 4
 - Table, 8
- Strings
 - Assembler, 51, 56
 - SYSTEM** Module, 16, 47
 - SYSTEM.7BJ**, 17
 - SYSTEM.OBJ**, 17
- Tokens, 50
- Trap Flag, 34
- TSR** Module, 73, 79
- TSR** Program, 73
 - Activating, 74
 - Cautions, 77
 - Deactivating, 78
 - Example, 76
 - Installing, 75
 - Terminating, 78
- TSR.DeInstall**, 78
- TSR.Install**, 75
- TSRCALC** Program, 79
- TSRCALC.MOD**, 77
- TxCOUNT**, 84
- TxFree**, 85
- Variables
 - Assembler, 57
- Warnings
 - OVLEXE**, 25
- Zero Flag, 35



Jensen & Partners UK Ltd.
63 Clerkenwell Road
London EC1M 5NP
England

Jensen & Partners International, Inc.
1101 San Antonio Road, Suite 301
Mountain View, CA 94043
USA