TMT Pascal

Multi-target Edition

Version 3.50 (Build 2.50)

Programmer's Reference

2000

Contents

About this guide

This document is a programmers's reference for the TMT Pascal language. It includes Win32 Programming, Compiler Directives, Run-time Error Codes and the PMODE/W DOS Extender.

Chapter 1

The TMT Pascal Language Description

1.1 Features

Overview

The TMT Pascal compiler is a fast compiler for the Pascal language. The compiler emits 32-bit code and supports many language extensions from Borland Pascal (BP), as well as more powerful extensions.

TMT Pascal brings new life to the 32-bit MS DOS applications and will help you to create your own Win32 and OS/2 applications.

Compilation targets

TMT Pascal allows easy building of the following targets:

– MSDOS 32-bit protected mode
– OS/2 presentation manager
– OS/2 console
– OS/2 full screen
– OS/2 DLL's
– Win32 GUI
– Win32 console
– Win32 DLL's

Language extensions

The TMT Pascal compiler supports a new enhanced dialect of the PASCAL language. This dialect fully covers the Borland Pascal language and has additional powerful extensions such as:

- C/C++ extensions support
- C++ and ADA-style comments
- Local declarations
- Multidimensional open arrays
- Operator overloading
- Unnamed procedural blocks
- MMX™ technology support

1.2 Implementation Issues

Memory Organization

The TMT Pascal compiler uses the TMTSTUB (based on WDOSX) and PMWSTUB (based on PMODE/W) extenders for a protected-mode program.

The segment registers are not used in protected mode. Instead all address space is separated into 4Kb pages.

You do not need to add a special _zero variable to get access to the physical addresses.

For example:

```
procedure clr_video(filler: char);
var
  i: Integer;
begin
  for i := 0 to 80 * 25 - 1 do
  Mem[$B8000 + i * 2] := filler;
end;
```

This procedure fills the video memory of the VGA adapter with the filler symbol.

Note that the linear address $B8000 is used as the physical address - not the segment address $B800.

Some other special variables are described in the SYSTEM unit. The *_psp* variable contains the logical 32-bit address of the PSP of the program, and the *_environ* variable contains the environment address.

Although you can access the interrupt vectors by using this method, we do not suggest doing this.

Also keep in mind that MS-DOS interrupt handlers use memory addresses in the 1st Mb of physical memory while your program and its data are loaded beyond the 1st Mb. The TMTSTUB intercepts and correctly handles some, but not all, calls to MS-DOS. Thus, if you are using *Intr* or *MsDos* calls, or call MS-DOS from the assembler, you will need to modify the code.

Absolute memory addressing Mem, MemW, MemL, and MemD pseudo-arrays may be used in BP-compatible manner:

```
var x: type absolute seg:offs
Mem[seg:offs]
```

Here the effective address is computed as seg*16+offs. The Ptr(seg, offs) function works similarly. The Seg(v) function still always returns 0.

These new functions should substantially simplify the conversion of the programs that use absolute addressing.

An example of using these functions can be found in file TMTPL\SAMPLES\MSDOS\FLAME\FLAME.PAS

See also: **PMODE/W API**

Calling Conventions

Calling conventions match those in Borland Pascal with the following differences:

- all parameters use 4 bytes on the stack, or a multiple of 4 (BP:2)
- all procedures must preserve the contents of registers ebx, ecx, edx, ds, and es!
- the direction bit should be cleared after the exit from a procedure, if it has been modified by the procedure.

To call external procedures written for different languages, TMT Pascal provides the **conv** operator. The **conv** operator should be used in the function (procedure) declaration to define a calling convention, which in turn will be used to call a declared function (procedure).

Syntax:

```
[function] conv conv_method FunctionName [Arguments] :
ReturnType;
```

Where *conv_method* is a constant, which defines the calling conversion to be used. The System units contains the following constants to define conventional method:

```
const
// Base calling conventions to construct any possible
convention
  arg_reverse    = [0];
  arg_proc_16    = [2];
  arg_noregsave  = [3];
  arg_no_drop_1  = [4];
  arg_no_drop_2  = [5];
  arg_no_drop_3  = arg_no_drop_1 + arg_no_drop_2;
  arg_no_drop_4  = [6];
  arg_no_drop_5  = arg_no_drop_1 + arg_no_drop_4;
  arg_no_drop_6  = arg_no_drop_2 + arg_no_drop_4;
  arg_no_drop_all= [4..6];
  arg_IO_test    = [8];
  arg_save_edi   = [9];
  arg_save_esi   = [10];
// Composite calling conventions
  arg_pascal     = arg_noregsave;
  arg_stdcall    = arg_reverse + arg_noregsave + arg_save_edi +
                   arg_save_esi;
  arg_cdecl      = arg_reverse + arg_no_drop_all;
  arg_os2        = arg_cdecl + arg_noregsave;
  arg_os2_16     = arg_proc_16 + arg_no_drop_all +
arg_noregsave;
```

The arg_pascal convention passes parameters from left to right; that is, the leftmost parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The arg_cdecl, arg_stdcall, arg_os2 and arg_os2_16 conventions pass parameters from right to left. For all conventions except arg_cdecl, the procedure or function removes parameters from the stack upon returning. With the arg_cdecl convention, the caller must remove parameters from the stack when the call returns. The register convention uses up to three CPU registers to pass parameters, whereas the other conventions always pass all parameters on the stack. The calling conventions are summarized in the following table.

Directive	Order	Cleanup	Registers
arg_pascal	Left-to-right	Function	No
arg_cdecl	Right-to-left	Caller	No
arg_stdcall	Right-to-left	Function	No

The *arg_pascal* and *arg_cdecl* conventions are mostly useful for calling routines in dynamic-link libraries written in C, C++, or other languages. The *arg_stdcall* convention is used for calling Windows API routines.

Limitations

1) Not implemented are *Mark* and *Release*.

2) The **Inline** operator is implemented in a partial form:

```
Inline(byte/byte/...);
```

References to variables/constants are not allowed.

3) Import of object modules does not support all 32-bit object formats. We recommend using **TASM** which is fully supported, except the usage of SEG addresses.

4) Complex type is not implemented.

5) Constants of **Extended** type are not supported.

6) The reserved word **Packed** has no effect (it is ignored) in TMT Pascal. Use **$OA: Objects and Structures Align Switch** to switch on or switch off objects and structures alignment.

1.3 Pascal Language Structure

TMT Pascal programs are to be written either with the TMT Pascal IDE editor or an editor of your choice. The source files created by your editor must be standard ASCII text. All characters within the range of 32 to 127 (decimal) are valid. Control characters (characters below 32 decimal) are treated as spaces.

Tokens and Identifiers

Contiguous characters in a source file, not including the space character (32), are called tokens. Tokens are separated by any number of spaces and control characters (in the range of 0 to 32 decimal). For instance in the following segment,

Writeln(‘Hello, World!’);

there are five tokens: the identifier *Writeln*, left and right parentheses, the semicolon and the string *‘Hello, World!’*. Programs are sequences of tokens that tell the compiler what code to generate. There are several different types of tokens; for instance, identifiers, reserved words, operators, and so on. Each type of token is explained below in this manual.

Identifiers are tokens that have a special meaning in TMT Pascal. Identifiers begin with a letter (A-Z or a-z) or underscore, and may contain letters, underscores, and digits (0-9). The maximum length of an identifier is 255 characters, however only the first 63 characters are significant. TMT Pascal is not case sensitive, therefore the identifiers *WriteLn*, *writeln*, and *WRITELN* are all identical. Reserved words, procedure names, and variables, are examples of identifiers.

Reserved Words

Reserved words are identifiers with a specific meaning in TMT Pascal. Their meaning cannot be changed or altered in any way. The following is a list of TMT Pascal reserved words:

```
and             goto            program
array           if              record
asm             implementation  repeat
begin           in              set
case            inherited       shl
const           inline          shr
constructor     interface       string
declare         label           then
destructor      library         to
div             mod             type
do              nil             unit
downto          not             until
else            object          uses
end             of              var
exports         or              virtual
file            overload        while
for             packed          with
function        procedure       xor
```

The following table shows TMT Pascal's standard directives. Directives are used only in contexts where user-defined identifiers can't occur. Unlike reserved words, you can redefine standard directives, but we advise you not to.

```
Absolute        Declare         os2call
Assembler       Export          name
Cdecl           external        virtual
Code            Forward         stdcall
Conv            Index
```

Operators and Delimiters

Operators and delimiters are tokens that also have special Pascal meanings. The following is a list of valid operators and delimiters along with their meanings:

token	Usage
`@`	Address operator
`^`	Pointer dereference operator
`+`	Addition or set union operator
`-`	Subtraction or set difference operator
`*`	Multiplication or set intersection operator
`/`	Real division
`div`	Integer Division
`mod`	Modulus
`()`	Parentheses
`[]`	Subscript delimiter, set constants
`=`	Assignment operator
`.`	Field selection operator
`,`	Separator
`..`	Range separator

:	Type separator or case separator
=	Equal operator
<	Less than operator
>	Greater than operator
<=	Less than or equal operator
>=	Greater than or equal operator
<>	Not equal operator
and	Logical AND
in	Set operator
not	Logical NOT
or	Logical OR
shl	Bit shift left replacing right side with 0's
shr	Bit shift right replacing left side with 0's
xor	Logical XOR

Operator Precedence

Operators allow for the manipulation of certain types of identifiers. For expressions with three or more operands (i.e. 5 *4 + 2), rules of precedence apply. The order of precedence for operators is as follows:

Operator	type
Unary Operators	**@, not**
Multiplying Operators	***, / , div, mod, and, shl, shr**
Adding Operators	**+, -, or, xor**
Relational Operators	**=, <>, <, >, <{}=, >{}= , in**

Operations are performed from left to right while operations of higher precedence are performed first. For more about operators see the chapter on **Expressions**.

Constants

A constant declaration (**const**) is an identifier that marks a value that can't change. TMT Pascal provides two standard types of constants:

- ### Integer and Real Number Constants

  Integer constants are values that can be represented in either decimal (base 10) or hexadecimal (base 16). A decimal number is a string of digits (0-9) that may be preceded with a plus or minus sign. A hexadecimal number is preceded by a dollar sign ($) followed by a string of digits and the characters A through F. The following are valid integer numbers:

  ```
  100   -255   100500   $FE   $ABCD
  ```

  Real constants are numbers that contain an integer portion, a fractional portion, and an exponent. Use real constants when the fraction of a number is necessary. The syntax for real constants is as follows:

  ```
  [+|-]  digits [.digits] [E [+|-] digits]
  ```

  The letter E represents the exponent part of the real number. Exponents are powers of ten. Both integer and real constants may not contain space characters. The following are valid real constants:

  ```
  1.0   -205.13   9019.31E100   40.71E-10
  ```

The current version of TMT Pascal compiler does not provide constants of Extended type.

- **String Constants**

String constants are strings of ASCII characters preceded by and followed by a single quote (‘). Use two single quotes (“) to represent a single quote within a string. A string may also be constructed with the number symbol # or the caret symbol ^. For more information see the definition of Character types. The maximum size of a string constant is 255 characters. The following are valid examples of string constants:

```
‘This is a string’
^G’A Bell will sound’
#13#11’New Line’
‘Where”s the program’
```

See also the chapter on **Constant Declarations**.

Program Comments

A good programmer knows that comments within a source file can be very helpful. Comments are delimited by «{« and «}» or «(*» and «*)». All comments are ignored by TMT Pascal during compilation. Comments cannot contain nested comments that use the same delimiters. Below are examples of traditional Pascal comments:

```
{ This is a comment }
(* Another comment *)
(* This comment is { nested } *)
{ Another (* nested *) comment }
(* An invalid (* comment *) *)
```

In addition to traditional comments, TMT Pascal supports C/C++ and Ada-style end-line comments. These begin with a double hyphen and span until the end of the line. For example:

```
/* This is C-style comment */
Space := ‘ ‘;  -- initialize filler char
FillChar(Ptr ($A0000), 64000, 0);  // clear VGA video memory
```

Remarks:

A comment that contains the dollar sign ($) immediately after opening {, (* or /* is a compiler directive. A mnemonic of the compiler command follows the $ character.

Starting from version 3.0, TMT Pascal does not support Ada-style comments by default. We are thinking about completely removing support for Ada-style comments in the future and recommend that you replace all Ada-style comments in your programs by traditional Pascal or C/C++ styled comments. Use the {$AC+} compiler switch, if you want to compile old sources with TMT Pascal 3.0 (see **$AC: Ada-Style Comments Switch**).

1.4 Types

A type defines the kinds and ranges of values that constants, variables, procedures, and functions may contain. Types also define the size of, as well as the operations on such identifiers.

TMT Pascal comes with a powerful set of predefined types and it is possible to define new types for constants and variables.

There are five basic type groups that are available under TMT Pascal. Each group contains types with similar properties. They are the following:

Scalar Types.
Scalar types consist of an ordered set of values. Scalar types include all ordinal types as well as real types. Characters are also of scalar type.

Ordinal Types.
Ordinal types are a subset of scalar types. Ordinal types include boolean, char, enumeration, and integer. Reals are not ordinal types.

Procedure Types.
Procedure types contain the address in memory of a procedure or function.

Pointer Types.
Pointer types store the address of a location in memory. Pointer types can be used to address dynamic variables.

Structured Types.
Structured types are types that contain several components. Each component can be accessed separately or the entire structure can be treated as a whole. Examples of structured types include strings, arrays, records, sets, files, and objects.

Boolean Types

A boolean type is an ordinal that can hold one of the two values: True, False. Expressions that evaluate to a logical “yes” or “no” are of boolean type. **if**, **while**, **repeat** and other control statements work with boolean expressions. The following code fragment,

```
  while not KeyPressed do;
```

causes program execution to pause until a key is pressed on the keyboard. *KeyPressed* is a procedure declared in the Crt unit. The expression **not** *KeyPressed* results in a boolean value that determines whether the **while** loop is executed. **not** performs a logical NOT on the boolean returned by *KeyPressed*. The constants True and False are declared in the System unit as boolean.

The evaluation model is controlled through the **$B** compiler directive. The default state is `{$B-},` so the compiler generates short-circuit evaluation code. In the `{$B+}` state, the compiler generates complete evaluation.

Character Types

Character types require one byte of storage. They may consist of any ASCII character, for instance, ‘A’ through ‘Z’, ‘0’ through ‘9’, or any control code. The code fragment below shows various ways to initialize character type variables.

```
var
  Number, Alpha, Bell, EofMarker: Char;
begin
  Number    := ‘5’;
  Alpha     := ‘a’;
  Bell      := ^G;
  EofMarker := #27;
end.
```

A character is normally delimited by two single quotes. However there are two other methods of representing characters as seen in the example above: the caret symbol (^) and the number symbol (#).

Use the caret to represent control codes—characters between 0 and 31 on the ASCII table. ^G stands for character number 7 because G is the seventh letter of the alphabet. When ^G is used during output, the computer's bell will sound.

Use # to represent any ASCII character. As in the example above, the end of file character, which is defined as character number 27 on the ASCII table, is assigned to the variable EofMarker. Note that #27 is the same as ^[.

Integer Types

Integer types may contain both positive and negative integer values. Integer values may range from -2,147,483,648 to 2,147,483,647 and other ranges are also supported. Each integer variable requires two bytes of storage. The following is a list of additional predefined integer types.

Type	Range	Size
Byte	0 to 255	1
ShortInt	-128 to 127	1
Integer	-32,768 to 32,767	2
SmallInt	-32,768 to 32,767	2
Word	0 to 65,535	2
LongInt	-2,147,483,648 to 2,147,483,647	4
DWORD	0 to 4,294,967,295	4
LongWord	0 to 4,294,967,295	4
Cardinal	0 to 4,294,967,295	4

On the Intel 386+ CPU's, operations performed with Longints (4 bytes) are faster than operations with integers (2 bytes). This is due to the fact that registers on the 32-bit processors are 32 bits wide.

Enumeration Types

Enumeration types are ordinals that represent a set of values specified by a list of identifiers. Enumeration types are defined as follows:

```
identifier [,identifier]
```

Each identifier is a constant of the new type. Identifiers in enumeration types are assigned values with the first equal to zero, the second equal to one, and so on. For instance, the following enumeration type contains the seven days of the week:

```
= (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

In type Week, Sun has the value of zero, Mon has the value of one, Tue has the value of two, and so on.

Enumerations are limited to 256 elements.

Subrange Types

Subrange types restrict the values allowed for a type. The subrange must consist of ordinal type constants and the components of the range must be of the same type. Subranges are defined as follows:

```
expression .. expression;
```

where the first expression is the lowest value of the range and the second expression is the highest value. The following are examples of subranges:

```
type
  Digits = '0'..'9';
  Values = 0..$F;
```

Real Types

Real data types contain integer values as well as a fractional portion. Also known as floating point numbers, each real type consists of a significant, the fractional part, and an exponent, which is a power of ten.

TMT Pascal follows the IEEE standard for floating point number representation. There are four real data types available under TMT Pascal: real, single, double, and extended.

Real Types.
Real types range from **2.9 x 10E-39** to **1.7 x 10E38** with 11 to 12 significant digits. Each requires 6 bytes for storage. The internal format of TMT Pascal's real type differs to Borland's.

Single Types.
Single types range from **1.5 x 10E-45** to **3.4 x 10E38** with 6 to 7 significant digits. Each requires 4 bytes for storage.

Double Types.
Double types range from **5.0 x 10E-324** to **1.7 x 10E308** with 15 to 16 significant digits. Each requires 8 bytes for storage.

Extended Types.
Extended types, the largest of all the real types, range from **1.9 x 10E-4951** to **1.1 x 10E4932** with 19 significant digits. Each requires 10 bytes for storage.

Pointer Types

Pointer types contain the address of an identifier or dynamically allocated memory. Pointer types require a double word (32 bits for storage). While in protected mode, pointers contain a 32-bit offset and the segment is assumed to be the data segment. In this way pointers are similar to the integer type Longint. To allocate dynamic variables see the chapter on Heap Management. A pointer type must point to specific type. Pointer types are defined as follows:

```
^Typename
```

The pointer type can be assigned the constant **nil**. When **nil** is assigned, the pointer does not refer to any location in memory. Pointer is a predefined pointer type that is untyped. Pointer is compatible with all other pointer types.

TMT Pascal supports the following arithmetical operations on pointers.

Examples:

```
p2 := p1 + 1000;
p1 := p2 - $FF;
inc(p1, 15);
dec(p2);
```

Where *p1* and *p2* are variables of Pointer Type.

Pointer Dereference

Dereferencing is used to refer to the object which a pointer type points to. Use

```
Variable^
```

to dereference a pointer type. It is important to note that pointer types must point to a specific memory location before they are dereferenced. Otherwise, the data pointed to is undefined.

Array Types

array types contain a sequence of components of a different type. Each component is referenced by an index which also has a specific type. Arrays are defined as follows:

```
array [Ordinalindex [,OrdinalIndex]] of Arraytype;
```

where *OrdinalIndex* is one of the following ordinal types: integer, char, enumeration, Boolean, or subrange. *Arraytype* can be of any type, including another array type. The following are examples of array types:

```
array [Boolean] of Char;
array [Char] of Integer;
array [1..255] of Double;
```

Subscripts are used to refer to a component of an array. Subscripting an array is specified as follows:

```
Arrayname [expression [,expression]];
```

where *expression* is of the same type as the index. *Expression* must also lie within the range of the index. A runtime error is generated if *expression* lies outside the index range and range checking is turned on, *$R+*.

String Types

Strings are arrays of characters. The maximum size of a string type is 255 characters. In the following example two string variables are declared, one with a specific size, the other with the maximum size of 255 characters.

```
S1: String[100];
S2: String;
```

The variable *S1* holds only the first 100 characters of a string. *S2* may contain up to 255 characters. TMT Pascal reserves one byte, which contains the size of the string. This size byte is reserved in the byte that precedes the first character of a string. For instance,

```
S2 := 'Hello World';
```

is represented in memory as

```
#11,Hello World
```

All string operations, as well as functions and procedures that return or modify strings, truncate strings that exceed 255 characters.

Set Types

Set types specify a subset of a set of values. The ordinal value of the set elements must range between 0 and 255.

- Characters
- Enumeration type
- Positive integer values
- Subrange of the above three
- Ordinal types

Each value in a set is defined by one bit, therefore each value is similar to a Boolean. The following are examples of sets:

```
type
  set of 0..7;
  set of '0'..'9';
  set of Char;
  set of Word;  - This cannot be handled by BP.
```

Record Types

Record types are structures that contain components of different types. Each component of a record is called a field. Variant sections are parts of records and can have multiple definitions. Record type are defined as follows:

```
record
   Fieldlist
end
```

where *Fieldlist* is defined as:

```
[[Fixedpart] | [Fixedpart;] [Variantpart]];
```

where *Fixedpart* is:

```
Field [; Field];
```

where *Field* is

```
identifier [,identifier] : Identifiertype;
```

Variantpart is defined as follows:

```
case
   [identifier:] Typename of Variant [;Variant];
```

where *Variant* is:

```
Caserange [,Caserange] : (Fieldlist);
```

where *Caserange* is:

```
expression [..expression];
```

With a proper understanding of TMT Pascal record types, very powerful types can be defined. The following are examples of record types:

```
type
  Coordinate = record
    x, y: Integer;
  end;
  Values = record
    case Way: Boolean of
      True:  (RValue : Extended);
      False: (IValue : Longint);
    end;
```

To reference a field of a record specify the record variable followed by a period (.) followed by the field name. The following refers to the fields of Coordinate declared above:

```
Coordinate.x
Coordinate.y
```

File Types

File types are structures that contain components of any type except another file type. File types are defined as follows:

```
File [of Componenttype];
```

If **of** is not specified and component types are not indicated then the file is untyped. Untyped files are used to access files regardless of their structure. Text file types refer to a file of ASCII characters grouped in lines. Text is a predefined type.

The record definitions used internally by TMT Pascal are also declared in the System unit.

TFileRec is used for both typed and untyped files.

```
type TFileRec = object

  magic   : ^TFileRec;
  name    : string;
  handle  : Longint;
  rec_len : Longint;
  state   : %flags;
  rd_proc,
  wr_proc : function (F: Longint; Buf: Pointer; Len: Longint;
var Act: Longint): Longint;

   procedure check_magic;
   procedure check_opened;
   procedure check_readable;
   procedure check_writeable;
   procedure io_error(code: Integer);
end;

PFileRec = ^TFileRec;
```

TTextRec is the internal format of a variable of type text.

```
type TTextRec = object(TFileRec)

  buffer  : array [0..63] of Char;
  index   : Longint;
  len_buf : Longint;
  max_buf : Longint;
  buf_adr : Pointer;
  function  Eof: Boolean;
```

```
   procedure init;
   procedure fill_buf;
   procedure fill_chr;
   procedure skip_spaces;
   procedure get_n_char(n: Integer);
end;

PTextRec = ^TTextRec;
```

Internal Type *%flags* is declared as:

```
type %flags = set of %file_state;
```

where %file_state is:

```
type %file_state =
(
    %file_readable,  //00h
    %file_writeable, //01h
    %file_opened,    //02h
    %file_assigned,  //03h
    %file_eof,       //04h
    %file_text,      //05h
    %file_file,      //06h
    %file_fileof,    //07h
    %file_tty,       //08h
    %file_special,   //09h
    %file_settextbuf //0Ah
);
```

Procedure Types

Procedure types contain the address of a procedure or function. Procedure types are followed by blocks of data and code and are defined as follows:

```
procedure identifier [(Parameterlist)]}
```

or

```
function identifier [(ParameterList)] : ReturnType}
```

ReturnType is the type of value returned by the function. *ParameterList* is defined as:

```
Parameter [;Parameter]}
```
where Parameter is:

```
[var] identifier [,identifier] : Typename}
```

var specifies a variable parameter. **var** parameters are passed by reference as opposed to being passed by value. The following are examples of procedure types:

```
procedure PrintAt(X, Y: Integer; S: String);
function  Max(Value1, Value2: Integer): Integer;
procedure GetDir(Driver: Byte; var S: String);
```

Note that unlike pointers, procedure types occupy 8 bytes. Besides the address of the procedure proper, the local frame is stored. This allows the forming of procedure types out of local procedures.

Object Types

Object types are similar to record types in that they contain components of different types. However, unlike records, objects may contain methods and be inherited. The full description of object types can be found elsewhere (see the **OOP Extensions** chapter.)

Type Compatibility

There are three levels of type compatibility. Each level along with its restrictions is listed below in order, from the most restrictive to the least restrictive.

Equivalent Types.
Two types are equivalent only if they are both defined from the same type declaration. That declaration must be one if the following:

- A named type declared by a type declaration.
- A predeclared type.
- An unnamed type used in a declaration.

Compatible Types.
Two types are compatible if one of the following apply:

- They are equivalent.
- One type is a subrange of another.
- Both are subranges of the same type.
- Both types are integers.
- Both types are reals.
- Both types are strings.
- One is a char or array of char and the other is a string.
- Both are set types and their base types are compatible.
- Both are arrays of char with the same length.

Assignable Types.
A type is assignable to another type if one of the following apply:

- Both types are compatible.
- Assignment of an integer type to a real type.
- Assignment of a char type to a string type.
- An array of characters less than 256 characters to a string.
- Both procedure types have the same parameters.

1.5 Declarations

The name of each identifier must be declared in your source code. By declaring an identifier to be of a particular type, such as a variable, or constant, you define its size and attributes.

Pascal is a block structured language. Each program, unit, procedure and function defines a block. Blocks can be nested creating blocks within blocks. The block structure effects the interpretation of identifiers such as constants, variables, types, and so on. Identifiers can have different meanings depending upon which block is referenced.

When an identifier is declared it is defined from the point of its declaration to the end of the inner most block that contains the declaration. This is the scope of the identifier. Redefinition of an identifier is not allowed within the same block in which it was declared. Only in nested blocks may an identifier be redefined. However in this case the new identifier does not refer to the old identifier. In fact the old identifier is hidden until the end of the nested block.

In addition to the traditional Pascal syntax, TMT Pascal also allows local block declarations. These are described below in the manual.

An identifier may not be referenced prior to its declaration, with one exception. A type name can be used as the base type of a pointer if the name is then declared in the type declaration that contains the reference. For instance:

```
type
  CoordPtr   = ^Coordinate;
  Coordinate = record;
    x, y     : Integer;
  end;
```

As you can see *Coordinate* is referenced prior to its declaration. The above declaration is valid due to the fact that *Coordinate* is declared in the same type declaration as *CoordPtr*.

There are several declarations possible under TMT Pascal. They include:

- Type Declarations
- Label Declarations
- Constant Declarations
- Variable Declarations
- Local Block Declarations

Type Declarations

Type declarations are preceded by the **Type** reserved word.

```
Type identifier = Identifiertype;
```

Identifier is the actual name of the new type you define. *Identifiertype* is the type of identifier. *Identifiertype* can be on of the following:

```
Another Type     String        Pointer      File
Subrange         array         set          function
Enumeration      record        procedure    Text
```

Some examples of type declarations are:

```
type
  Float        = Extended;
  Int          = Integer;
  Filename     = array [0..8] of Char;
  Fnameptr     = ^Filename;
  Seasons      = (Winter, Spring, Summer, Autumn);
```

Label Declarations

TMT Pascal statements may be labeled with one of two types of labels. A label can either be a positive integer number (0...2147483647) or an identifier. Before using labels in your code, a label must be declared. A label declaration is preceded by the **label** reserved word.

```
label identifier [,identifier];
```

Constant Declarations

There are two types of constants and TMT Pascal interprets them in different ways.

Constants that are declared without a type may not be changed in the program. Constants that are typed are the same as variables (described below), except that they contain an initial value. These constants may be changed in the program. A constant declaration is preceded by the **const** reserved word.

```
const identifier [:IdentifierType] = expression;
```

Identifier is the actual name of the constant defined. *IdentifierType*, which is optional, specifies the type of the constant. Again, if a type is specified, the constant is the same as a variable with an initial value. *Expression* is assigned to the constant and must be evaluated at compile time. The following are untyped constants and may not be modified:

```
const
  Digit   = '0'..'9';
  MaxSize = 100;
  Msg     = 'This is a string constant';
```

Typed constants may be of any type except for file, procedure, or function. Some examples of typed constants are:

```
type
  Coordinate = record
    x,y: Integer;
  end;

const
  Originpos: Coordinate = (x:0; y:0);
  Name     : String = 'Hello World!';
  StrSize  : Integer = 100;
  Ary      : array[False..True] of Byte = (10,15);
```

See also: **Integer and Real Number Constants, String Constants**

Variable Declarations

Variables store data during program execution. A variable declaration is preceded by the **var** reserved word.

```
var identifier [,identifier]: IdentifierType;
```

Identifier is the actual name of the variable defined. *IdentifierType* specifies the type of the variable. Variables, unlike constants, are not initialized. Their content prior to initialization is undefined. It is possible to specify the absolute address of a variable. Following the *Identifiertype* specify the Absolute reserved word.

```
var
  identifier [,identifier]: IdentifierType absolute address;
```

Address may be either an identifier or an integer number indicating an offset. If *Address* is an identifier then TMT Pascal computes it's offset. **absolute** maps the variable to the address following the **absolute** statement. This identifier must be declared prior to using **absolute**. If a variable is the absolute of another, both address the same data however the types may be different. The following are examples of variable declarations.

```
const
 BuffSize = 900000;  // 900K
var
 HugeBuff : array [0..BuffSize] of Char;
 i,j,k,l  : Integer;
```

```
 Buffptr  : Longint;
 p        : Pointer absolute Buffptr;
 Alpha    : Char;
```

The absolute may refer to fields of records and objects. Also, the address of a global record/object field can be used within the initialization of typed constants. Furthermore one can use recursive initialization:

```
type rec = record
  next:    ^rec;
  buffer:  array [1..10] of char;
  buf_adr: pointer;
end;
const cyclic: rec = (next: @cyclic; buf_adr: @cyclic.buffer);
```

Local Block Declarations

It is very often necessary to declare a local variable with a short life-span. One has to do this at the declaration part of a program or in the procedure body. This is not always convenient, especially if there is a huge program with a complicated algorithm. For that case a special construction has been added in TMT Pascal. It is called a Local or Nested Block. Such a block is an ordinary compound statement which begins with the new reserved word **declare** and consists of two parts - declaration and execution:

```
declare
  <declaration part>
begin
  <execution part>
end;
```

This statement can be used in any place where a structured statement can be placed.

Example:

```
program DeclDemo;
var
  b: Integer;
begin
  declare  // first local block
    var
      a: Integer;
    procedure pr_int(a: Integer);
     var
       i: Integer;
     begin
       for i := 1 to a do
       declare  //second local block
       var
         k: Integer;
       begin
         k := a div i;
         Writeln(a, ' div ', i, ' =  ', k);
       end;
     end;
  begin
     a := 1;
     Writeln(a);
```

```
    b := 10;
    pr_int(b);
  end
end.
```

This example contains two local blocks: one of them in the program body and another in the routine body. The first local block declares variable *a* and procedure *pr_int*, the second declares one local variable *k*. It should be understood that the scope of 'a' and *pr_int* is the interior of the "first local block," and the scope of *k* is the interior of the "second local block."

1.6 Expressions

Expressions are constructs made up of operators and operands. Expressions work with existing data and return new data. In TMT Pascal there exist two types of operations, unary and binary. Unary operations work with one operand and binary operations work with two. Regardless of the operator, operands may be constants, variables, data returned by another operator, or data returned by a function call. Operators can be grouped according to the types they operate on. There are five groups of operators: integer, real, Boolean, set, and relational.

Here is a description of the operator groups:

- Arithmetic Operators
- Boolean Operators
- Set Operators
- Relational Operators
- Typecasts
- Operator Precedence

Arithmetic Operators

Standard arithmetic operators listed below.

Operator	Operation
@	Pointer formation
+	Unary sign identity
-	Unary sign negation
+	Addition
-	Subtraction
*	Multiplication
Div	Integer division
/	Real division
Mod	Integer remainder
And	Logical AND
Xor	Logical XOR
Not	Logical NOT
Or	Logical OR
Shl	Shift bits left
Shr	Shift bits right

During binary operations both operands must be of compatible type. If the operands are of compatible type then the operation results in the same type of the operand. If the types are different then the result is the larger type.

For integer operations, operands are converted to Longint and results are of the same type as the destination type. Longint or 32 bit operations are faster on the 80386 and 80486.

During real operations, operands are converted to extended type and results are of the same type as the destination.

Boolean Operators

Boolean operators include logical AND, NOT, OR, and XOR. The operation of each is summarized below:

Operator	Logical Operation
AND	Conjunction
NOT	Negation
OR	Disjunction
XOR	Exclusive Disjunction

Boolean expressions that evaluate to True return a value of one. Boolean expressions that are False result in a zero value.

Set Operators

Set operators are defined as follows:

Operator	Meaning	Operation
+	Union	Yields elements in either A or B
-	Difference	Yields elements in A but not in B
*	Intersection	Yields elements in both A and B

Relational Operators

Relational operators perform arithmetic, literal, and set comparisons. All relational operations result in a Boolean type. Relational operators include:

Operator	Meaning	Applicable types
=	Equal	integers, reals, booleans, chars, enumerations, strings, sets, pointers
<	Less than	integers, reals, boolean, char, enumerations, strings, pointers
>	Greater than	integers, reals, boolean, chars, enumerations, strings, pointers
<=	Less than or equal, set inclusion	integers, reals, booleans, chars, enumerations, strings, pointers
>=	Greater than or equal, set inclusion	integers, reals, booleans, chars, enumerations, strings, pointers
<>	Not equal	integers, reals, booleans, chars, enumerations, strings, sets, pointers
In	Membership	A set type on the right and the set's base type on the left

Typecasts

Typecasts allow operands of one type to be converted to another type. Typecasts are allowed on either values or variables. Typecasts on values are restricted to ordinal and pointer types. The only restriction on typecasts on variables is that the sizes of both types must be the same. The following are examples of typecasts.

```
Integer('0')
Boolean(1)
Wordptr(@BuffPtr)
Char(27)
Longint(@BuffPtr)}
```

Operator Precedence

For expressions with three or more operands (i.e. 2 - 244 / 4), rules of precedence apply. The order of precedence for operators is listed from highest to lowest:

Operator Type	Operator
Unary Operators	**@,Not**
Multiplying Operators	***,/,Div,Mod,And,Shl,Shr**
Adding Operators	**+,-,Or,Xor**
Relational Operators	**=,<>,<,>,<=,>=,In**

Operations are performed from left to right while operations of higher precedence are performed first. For instance, the following expression:

```
7 + 4 * 2
```

is not the same as:

```
(7 + 4) * 2
```

Since multiplication has a higher precedence than addition, multiplication is performed first followed by addition. Use parenthesis to separate operations that you want to be performed first.

1.7 Statements

A statement indicates the action a program performs. Statements are separated by semicolons (;). Statements may be preceded by a label which consists either of an identifier or an unsigned integer constant.

Assigments

An assignment assigns a value to a variable. An assignment takes the following form:

```
variable := expression;
```

where the value returned by expression is stored in variable. The type of the value returned by expression must be compatible with the type of variable. If variable appears in expression, its value is the value prior to the assignment. The following are examples of assignments:

```
const
  Letter    = 'A';
var
```

```
  Alpha     : Char;
  Value, i  : Integer;
  l         : Longint;
  s         : String;
begin
  Alpha     := Letter;
  s         := 'A string variable';
  Value     := $643F;
  i         := 2675;
  l         := 200 + (Value * i);
end.
```

Compound Statements

Compound statements are comprised of single statements preceded by **begin** and followed by **end**. Compound statements take the following form:

```
begin
      [statement [; statement]]
end
```

Compound statements allow one to place two or more statements wherever a statement is called for within another statement.

Case Statement

The **case** statement selects from a list of statements basing it's decision on the value of an expression. **case** statements take the following form:

```
case expression of
  Selector : statement
  [else statement]
end;
```

where expression is a value of ordinal type. The **case** expression value is matched against each *Selector*. If a match exists the statement following the matching *Selector* is executed. *Control* is then transferred out of the **case**. If no *Selector* matches the **case** expression then control is passed to an optional else clause. *Selector* must evaluate to a constant at compile time and is defined as:

```
expression [..expression] [,expression [..expression]]}
```

if .. is specified followed by another expression the **case** applies to the entire range between the first expression and the second expression. The following is an example of the **case** statement:

```
case Int of
  5       : WriteLn('Int is 5');
  7..12,15: WriteLn('Between 7..12 or 15');
  else      begin
              WriteLn('Undefined.');
              GetNextInt;
            end;
end;
```

Performance for large **case** statements improves if the most common subcases are listed first.

For Statement

The **for** statement allows for repetitive execution of one or more statements. **for** executes a loop for a predetermined number of iterations. **for** statements take the following form:

```
for variable := expression to | downto expression
  do statement
```

where variable must be of ordinal type. The first expression following variable is the initial value that is assigned to variable and the second expression is the limit on the range of values assigned to variable. Both expressions must be of compatible type.

The **to** or **downto** clause specify specifies whether a variable is incremented or decremented after each iteration of the loop. If **to** is specified, the variable is incremented until it hits the limit of the second expression. **downto** decrements variable until it reaches the lower limit of the second expression.

The following are examples of **for** loops:

```
for i := 1 to 100 do
begin
  WriteLn(i);
  Intarray[i] := i + 4;
end;

for x := 5 downto 2 do
  WriteLn(x);
```

A **for** loop is not executed if the first expression is greater than or less then the second expression depending upon whether a **to** or **downto** was specified. For instance the following **for** loop is not executed:

```
for i := 5 to 4 do

   WriteLn('Will never output this string!');
```

For a loop, the index variable must either be global or local to the procedure to which it belongs.

Goto Statement

As mentioned above statements may be preceded by labels. The **goto** statement transfers control to a specific label. The format of a **goto** statement is as follows:

```
goto label;
```

where label has been previously declared in the current block. The following is an example of the **goto** statement:

```
label
  GotoLoop;
var
  i: Integer;
begin
  i := 1;
GotoLoop:
  WriteLn(i);
  Inc(i,2);
  if i < 100 then
    goto GotoLoop;
end.
```

If Statement

The If statement conditionally executes one of two statements based on the value of an expression. If statements take the following form:

```
if expression then
      statement
[else statement]
```

where *expression* evaluates to a Boolean value. If *expression* results in True then the statement following the reserved word **then** is executed. Control is then transferred to the first *statement* outside the **if** statement.

If *expression* evaluates to False and an **else** clause is specified then the statement following **else** is executed. If no **else** clause exists and *expression* is False then the **if** statement is passed over. The following is an example of an **if** statement.

```
if Flag then
      WriteLn('Expression is True')
else
      WriteLn('Expression is False');
```

The end of an **if** statement is indicated by a semicolon (;). In the above example, if *Flag* is a constant then TMT Pascal optimizes code generation and automatically eliminates code that is never executed.

InLine Statement

The **inline** clause is used to define a short machine language routine. **inline** procedures are treated as macros rather than procedure calls and are therefore extremely efficient. It is recommended that you have thorough knowledge of 32 bit assembler before writing machine code macros.

```
function IsLower(Ch:Char):Boolean;
  inline(
         $58/           { pop     eax    }
         $3C/$61/       { cmp     al,'a' }
         $0F/$90/$C4/   { setge   ah     }
         $3C/$7A/       { cmp     al,'z' }
         $0F/$9E/$C0/   { setle   al     }
         $22/$E0)       { and     al,ah  }
```

Notice the use of new 80386 and 80486, Pentium, AMD 3DNow! and Intel MMX instructions. For more information about CPU extensions refer to your Intel™ and/or AMD™ reference manuals.

Repeat Statement

The **repeat** statement, much like the **for** statement, executes one or more statements in a loop. Unlike a **for** statement where the loop condition is tested prior to each iteration, a **repeat** statement condition is tested after each iteration. Therefore a **repeat** loop is executed at least once. **repeat** takes the following form:

```
repeat
statement [; statement]
until expression;
```

where the **repeat** loop executes until the expression evaluates to the boolean value of True. When the expression is False, the loop is executed again. The following is an example of the **repeat** statement:

```
repeat {Do nothing} until KeyPressed;
```

While Statement

The While statement executes one or more statements in a loop. **while** statements take the following form:

```
while expression do
statement
```

where expression evaluates to a Boolean type. A **while** loop executes until expression evaluates to False. **when** False, control is transferred to the first statement outside the **while** loop.

With Statement

The **with** statement allows one to refer to the fields of a record type as if they were independent variables. **with** takes the following form:

```
with variable [, variable] do
   statement
```

where variable refers to a record type. *Statement* may refer to the fields of variable without specifying the variable name. The following is an example of the **with** statement:

```
type
  ScreenString = record
  x, y: Integer;
  str : String;
end;
var
  ScrnSay : ScreenString;
begin
  with ScrnSay do
  begin
      x  := 5;
      x  := 5;
      y  := 10;
      str:= 'Hello World!';
  end;
end.
```

Mem, MemW, MemL, and MemD

TMT Pascal implements four predefined arrays to directly access memory:

Mem, MemW, MemL, and MemD.

- Each component of Mem is a Byte
- Each component of MemW is a Word
- Each component of MemL is a Longint.
- Each component of MemD is a DWORD.

See also: **Memory Organization**

Port, PortW and PortD

TMT Pascal implements three predefined arrays to directly access 80x86 CPU data ports:

Port, PortW, and PortD.

Port, PortW, and PortD are one-dimensional arrays, and each element represents a data port whose port address corresponds to its index.

- Each component of Port is a Byte
- Each component of PortW is a Word
- Each component of PortD is a DWORD.

When a value is assigned to a component of Port, PortW or PortD, the value is output to the selected port. When a component of Port, PortW or PortD is referenced in an expression, its value is input from the selected port.

1.8 Programs and Units

TMT Pascal source files contain units, programs, or both. A unit is a collection of procedures, functions, and data that is accessible to other programs or units. Units aid in the modular design of applications and are similar to libraries. Units may not be executed directly. A program consists of one or more procedures or functions. The main procedure of a program is executed during runtime.

Units

Units can be compiled separately and take the following form:

```
unit Unitname;
interface
  [Declaration]
implementation
  [Declaration]
[
  begin
  [statement [; statement]]
]
end.
```

Unitname is the name of the unit. This is the same name that you will use in programs to reference the unit. There are three sections within each unit.

Interface Section

The **interface** section contains declarations of types, constants, variables, procedures, and functions that are public and accessible to other programs and units. When declaring procedures and functions in the Interface section, only the procedure header is required. These declarations are similar to using the **forward** clause that tells TMT Pascal that the complete declaration is further ahead in the program. The entire procedure declaration is done in the **implementation** section. Local variables and procedures that need not be accessible outside of the unit may be declared in the **implementation** section.

Implementation Section

The **implementation** section contains local types, constants, variables, labels, procedures and functions. Procedures and functions are local to the unit unless their header is also declared in the **interface** section. The **implementation** section contains complete procedure and function declarations.

Unit Initialization

The initialization section starts immediately after the **begin** statement. This code block is executed by the main program that uses the unit. It is executed prior to the main code block.

Each unit is terminated by the **end** statement followed by a period. The reserved words Interface and Implementation must be specified in a unit. The initialization section is optional.

In Borland Pascal, private procedures are compiled as near while public procedures are far. Therefore private procedures are more efficient. In TMT Pascal both private and public procedures are near and equally efficient.

Programs

Programs require a different format from units. The general format takes the following form:

```
[program identifier;]
[uses Unitname [, Unitname]]
[Declaration]
begin
  statement [; statement]
end.
```

The identifier following the **program** statement declares the name of the program. Program files are terminated by the **end** statement followed by a period (.).

The **uses** statement tells TMT Pascal which units it uses. *Unitnames* listed after the **uses** statement are loaded by TMT Pascal. Procedures and variables referenced by the program are linked into the executable generated. All types, constants, variables, and functions declared in the Interface section of units are accessible to the program.

All text beyond the final **end** statement in either a unit or program is ignored by TMT Pascal.

In TMT Pascal the main program may contain **interface** and **implementation** parts as well. This allows access to the variables of the main program from other modules:

```
// Test Program
program Test;

interface
  var global: Integer;

implementation
uses UnitTest;

begin
  UnitTest.Write;
end.

// Test Unit
unit UnitTest;
```

```
interface
  procedure Write_global;

implementation
uses Test;
  procedure Write_global;
  begin
    Write(test.global);
  end;
end.
```

The name of the file that contains the text of the main program or unit must be identical with the name that follows the keyword **program**.

1.9 Dynamic-Link Libraries (DLL's)

Targets: OS/2, Win32

About DLL's

In Microsoft® Windows® and IBM © OS/2 © operating systems, dynamic-link libraries (DLL) are modules that contain functions and data. A DLL is loaded at runtime by its calling modules (.EXE or DLL). When a DLL is loaded, it is mapped into the address space of the calling process.

Dynamic linking has the following advantages over static linking:

- Processes that load a DLL at the same base address can use a single DLL simultaneously, sharing a single copy of the DLL code in physical memory. Doing this saves memory and reduces swapping.
- When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments, calling conventions, and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
- A DLL can provide after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was initially shipped.
- Programs written in different programming languages can call the same DLL function as long as the programs follow the same calling convention that the function uses. The calling convention (such as C, Pascal, or standard call) controls the order in which the calling function must push the arguments onto the stack, whether the function or the calling function is responsible for cleaning up the stack, and whether any arguments are passed in registers. For more information, see the documentation included with your compiler.

A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module. The system terminates processes using load-time dynamic linking if they require a DLL that is not found at process startup and gives an error message to the user. The system does not terminate a process using run-time dynamic linking in this situation, but functions exported by the DLL are not available to the program.

DLLs can define two kinds of functions: exported and internal. The exported functions can be called by other modules. Internal functions can only be called from within the DLL where

they are defined. Although DLLs can export data, such data is usually used only by its functions.

DLLs provide a way to modularize applications so that functionality can be updated and reused more easilly. They also help reduce memory overhead when several applications use the same functionality at the same time, because although each application gets its own copy of the data, they can share the code.

TMT Pascal Multitarget support DLLs for Win32 and OS/2 compilation targets. DLLs are not supported for MS-DOS protected mode target.

Using DLLs

TMT Pascal provides two ways to import procedures and functions:

- by new name
- by index

Example:
This external declaration imports the function ExitProcess from the system DLL called KERNEL32 (the Windows 32 kernel):

```
procedure ExitProcess conv arg_stdcall (uExitCode: DWORD);
external kernel32dll name 'ExitProcess';
```

Example:
This example program imports ArcCos and ArcSin functions from the DLL called ARCs (see **Writing DLLs**):

```
program TestDLL;

uses Strings;

const
  ARCs = 'arcs.dll';

// import by name
{$ifdef __WIN32__}
function ArcCos conv arg_stdcall (X: Extended): Extended;
  external ARCs name 'ArcCos';
{$else}
function ArcCos conv arg_os2 (X: Extended): Extended;
  external ARCs name 'ArcCos';
{$endif}

// import by index
{$ifdef __WIN32__}
function ArcSin conv arg_stdcall (X: Extended): Extended;
  external ARCs index 1;
{$else}
function ArcSin conv arg_os2 (X: Extended): Extended;
  external ARCs index 1;
{$endif}

var Arg: Extended;

begin
  repeat
    Write('Argument ? ');
```

```
    Readln(Arg);
    if (Arg < -1) or (Arg > 1) then
      Writeln('Argument must be in range: [-1..1]');
  until (Arg >= -1) and (Arg <= 1);
  Writeln('ArcCos(', Fls(Arg), ') = ', Fls(ArcCos(Arg)));
  Writeln('ArcSin(', Fls(Arg), ') = ', Fls(ArcSin(Arg)));
end.
```

Writing DLLs

The structure of a TMT Pascal DLL is identical to that of a program, except that a DLL starts with a library header (**Library**) instead of a program header (**Program**).

All procedures and functions which are to be exported by a DLL, must be compiled with the export procedure directive.

If you want your DLL to be available to applications written in other languages, it's safest to specify the *arg_stdcall* calling convention in the declarations of exported functions. Other languages may not support TMT Pascal's default register calling convention.

Example:

```
// This implements a very simple DLL with two exported
functions:

library ARCs;

// The export procedure directive prepares ArcCos
// and ArcSin for exporting

uses Math;
{$ifdef __WIN32__}
function ArcCos conv arg_stdcall (X: Extended): Extended;
{$else}
function ArcCos conv arg_os2 (X: Extended): Extended;
{$endif}
begin
  Result := RadToDeg(ArcTan2(Sqrt(1 - X * X), X));
end;

{$ifdef __WIN32__}
function ArcSin conv arg_stdcall ( X: Extended): Extended;
{$else}
function ArcSin conv arg_os2 ( X: Extended): Extended;
{$endif}
begin
  Result := RadToDeg(ArcTan2(X, Sqrt(1 - X * X)));
end;

// The exports clause actually exports the two routines,
// supplying an optional ordinal number for each of them

exports
  ArcCos name 'ArcCos',        // export by name
  ArcSin index 1;              // export by index

begin
  // Do nothing
end.
```

Global variables in DLLs

Global variables declared in a DLL cannot be imported by a TMT Pascal application. A DLL can be used by several applications at once, but each application has a copy of the DLL in its own process space, with its own set of global variables. For multiple DLLs or multiple instances of a DLL to share memory, they must use memory-mapped files. Refer to the **Windows API** documentation for further information.

Import Units

You can place declarations of imported procedures and functions directly in the program that imports them. They are usually grouped together in an "import unit" that contains declarations for all procedures and functions in a DLL, along with any constants and types required to interface with the DLL. For instance, the **Windows** and **OS2MAPI** are import units.

Of course, import units are not a requirement of the DLL interface, but they do simplify maintenance of projects that use multiple DLLs. Also, when the associated DLL is modified, only the import unit needs updating to reflect the changes.

1.10 Procedures and Functions

Procedures are a sequence of instructions that are separate from the main code block. Functions are procedures that return a value. Other than this difference, both procedures and functions are the same.

Procedures are blocks of code that are called from one or more places throughout a program. Procedures make source code more readable and reduce the size of the executable because repetitive blocks of code are replaced with a call to a procedure. Both procedures and functions accept parameters. Parameters allow the calling routine to communicate with a procedure.

Parameters can be passed by value or by reference or by constant reference.

If passed by value, only the value of the parameter is passed and the procedure has no access to the actual variable. One can modify the value parameter. It will have an effect only inside of the procedure body and will not change the actual variable.

If passed by reference, also known as **var** parameters, an address of the memory location containing the value is passed thus making it possible to modify the variable.

If passed by constant reference, also known as **const** parameters, an address of the memory location containing the value is passed but the compiler does not allow one to modify a constant parameter and does not allow passing one as an actual variable parameter to another procedure or function.

Procedures and Functions Declaration

Procedures and functions take the following form:

```
procedure identifier [(Parameterlist)];
```

or

```
function identifier [(Parameterlist)] : ReturnType;
```

ReturnType is the type of the value returned by the function. *Parameterlist* is defined as:

```
Parameter [;Parameter];
```

where *Parameter* is:

```
[var] identifier [,identifier] [: TypeName];
```

or

```
[const] identifier [,identifier] [: TypeName];
```

var specifies a variable parameter. **const** specifies a constant parameters. **var** and **const** parameters are passed by reference as opposed to being passed **var** and **const** parameters are passed by reference as opposed to being passed by value.

The body of a procedure or function takes the following form:

```
[Declarations]
begin
  statement [; statement]
end;
```

Types, labels, constants, and variables declared in the declaration section prior to the **begin** statement are local variables. Space for these variables is allocated only when the procedure is called. Like all variable declarations their data is undefined until initialized. TMT Pascal procedures and functions may be called recursively.

All identifiers must be declared prior to being referenced. The same rule applies to procedures and functions.

Forward Declaration

The **forward** clause is used to define a procedure prior to its complete declaration. **forward** tells TMT Pascal that the declaration is further ahead in the program. A **forward** procedure declaration takes the following form:

```
Procedureheader; forward;
```

External Declaration

The External clause is used to define a procedure that is linked in from an assembly object.

Example:

```
;----------------- [vga.asm] ---------------
IDEAL
P386
MODEL FLAT,PASCAL

CODESEG

GLOBAL  SETVIDEOMODE: PROC
PROC    SETVIDEOMODE USES EAX, MODE: WORD
        MOV     AX,  [MODE]
        INT     10H
        RET
ENDP    SETVIDEOMODE

GLOBAL  CLEARVGA: PROC
```

```
PROC    CLEARVGA  USES ECX, COLOR: BYTE
        MOV     EDI, 0A0000H
        MOV     AL,  [COLOR]
        MOV     AH,  AL
        MOV     ECX, EAX
        SHL     EAX, 16
        MOV     AX, CX
        MOV     ECX, 64000/4
        CLD
        REP     STOSD
        RET
ENDP    CLEARVGA

END
;-----------------------------------------


//////////////// [Test.pas] ///////////////
program Test;

{$ifndef __DOS__}
This program can not be compiled for OS/2 or Win32
{$endif}

uses CRT;

{$l vga}            // include vga.obj file

procedure SetVideoMode(Mode: Word); external;
procedure ClearVGA(Color: Byte); external;

begin
  SetVideoMode($13); // setup VGA/MCGA mode 320x200
  ClearVGA(10);      // fill screen with green color
  ReadKey;           // wait for key hit
  ClearVGA(15);      // fill screen with white color
  ReadKey;           // wait for key hit
  ClearVGA(0);       // fill screen with black color
  ReadKey;           // wait for key hit
  SetVideoMode($03); // setup VGA text mode 80x25
end.
//////////////////////////////////////////
```

See also: **Dynamic-Link Libraries (DLL's)**

Interrupt Procedure

In TMT Pascal, the Interrupt clause defines a procedure that is to be used as an interrupt handler.

The parameters of an interrupt procedure are the CPU registers. The following is the order of the CPU registers: EFLAGS, CS, EIP, EAX, EBX, ECX, EDX, ESI, EDI, DS, ES, EBP. If these register variables are assigned a new value, upon completion of the interrupt the new values will be restored onto the actual CPU registers.

Declaration:

```
procedure IntProc(used registers); interrupt;
```

An example below shows you a simple method of working with interrupt-handlers.

```
program Timer1;
uses Dos, Crt;
var
  Int1CSave: FarPointer;
  Time: LongInt;

// TimerHandler
procedure TimerHandler; Interrupt;
var
  StoreX, StoreY: Word;
begin
  Inc(time);
  StoreX:= WhereX;
  StoreY:= WhereY;
  GotoXY(1,1);
  Write(time);
  GotoXY(StoreX, StoreY);
  Port[$20] := $20;
end;
begin
  ClrScr;
  Time := 0;
  GetIntVec($1C, Int1CSave);
  SetIntVec($1C, @TimerHandler);
  Writeln;
  Writeln('Type something and press "ENTER" to exit');
  Readln;
  SetIntVec($1C, Int1CSave);
end.
```

When using Intr() and MsDos(), keep in mind that the DOS interrupt handlers can deal only with the addresses from the 1st megabyte of memory.

Procedural Value

TMT Pascal has a notion of a procedural value. It gives an opportunity to use a procedure or function in a program as a usual simple type object such as enumerate type or subrange type. One can declare a variable of the procedural type, make an assignment to it, and invoke the procedure body from it.

The procedural value implemented in the TMT Pascal occupies 8 bytes of memory and consists of two parts: the entry point to the routine and the reference to the local environment of the routine (known as a routine base). The format of a procedural value is the following:

```
0   +-----------------------+
    ¦ The entry point       ¦
4   +-----------------------¦
    ¦ The local environment ¦
8   +-----------------------+
```

The first part is needed for calling the routine. The second part is used to access the routine variables.

Such format of the procedural value is incompatible with the Borland Pascal format which has only the entry point.

Furthermore, the stack frame structure and parameter passing conventions differ from those in Borland Pascal.

Thus the approach used in TVision and CLassLib for writing iterations cannot be used. However, we offer this correct and reliable (and more standard) way:

```
type list = object
  next: ^list;
  procedure for_all(procedure body(var v));
end;
procedure list.for_all;
var
  p: ^list;
begin
  p := @self;
  repeat
    body(p);
    p := p^.next;
  end;
end;
...
type int_list = object(list)
  value: integer;
  function first_positive: ^int_list;
end;
function int_list.first_positive;
label OK;
var
  res: ^int_list;
procedure do_item(var v);
begin
  if int_list (v).value > 0 then
  begin
    res := @v;
    goto OK;
  end
end;
begin
  res := nil;
  for_all(do_item);
OK:
  first_positive := res;
end;
...
```

The procedural value from a method or object can be obtained by selecting the method from some object value (not from a type). The parameters of this procedural value must match the parameters of the method. The invocation of such a procedural value is an invocation of the corresponding method of the object. The reference to the object is transferred through the base of the procedural value.

You can use only global procedural values to initialize a type constant.

Procedural values may be used only while the environment where they were formed is still in existence. Thus,

– for local procedures—until the exit from the block, in which they are described;
– for methods—while the underlying object still exists.

Using Statement as Procedure

With TMT Pascal you can use any statement as a procedure body, except for the assignment and procedure calls.

The *RESULT* variable in the body of such functions denotes the variable that contains the return value. The *RESULT* is of the function return type and may be used as a variable without any restrictions.

With TMT Pascal you can enter the procedure body directly as a procedure parameter. The procedure or function header (if not specified) takes the procedural parameter type. If the procedure header is specified, the procedure name is omitted.

Example:

```
function Integral(function f(a: Real):Real; low, high, step:
Real): Real;
begin ... end;
...
Writeln(integral (
  function(x: Real): Real; begin Result := sqrt(x) end, 0, 10,
0.1));
Writeln(integral(begin Result := sqrt(a) end, 0, 10, 0.1));
Writeln(integral(
               function;        // function keyword needed
               var x: Real;     // for local declaration
               begin x := sqrt(a); Result := x end, 0, 10, 0.1)
              );
Writeln(integral(
                declare;      // other way
                  var x: Real;// for local variable declaration
                begin
                  x := sqrt(a);
                  Result := x
                end, 0, 10, 0.1)
               );
```

TMT Pascal allows an exit from a local procedure to the one that contains it. This feature is listed in the Pascal's ANSI standard but not realized in Borland Pascal. Together with procedural values, this is very useful for error handling:

```
program test;
var
  on_eof: procedure;
  function read_char: char;
var
  c: char;
begin
  if EOF(Input) then on_eof;
  Read(c);
  Read_char := c;
end;
procedure p;
label eof_reached;
  procedure go_eof; begin goto eof_reached; end;
  begin
    on_eof := go_eof;
    while True do Write(read_char);
    Eof_reached:
    Writeln('*** EOF ***');
    on_eof := nil;
  end;
begin
    p;
end.
```

break and **continue** operators cannot be used to exit from a procedure. Use g**oto** instead.

Example:

```
{ incorrect example }
for i := 1 to 10 do
Writeln (
  integral(            // from previous example
           if a < 0
           then break  // incorrect
           else result := sqrt (a),
           i, i + 1, 0.01)
          );
{ correct example }
declare
  label L;
begin
  for i := 1 to 10 do Writeln (
  integral(
           if a < 0
           then goto L  // correct
           else result := sqrt (a), i, i + 1, 0.01)
          );
```

Functions may return any values of any type, including structures and arrays.

1.11 OOP Extensions

TMT Pascal implements object oriented programming (OOP) extensions similar to the OOP extensions in Borland Pascal. This sections describes the applicable syntax of OOP.

Object

An object is a structure that consists of fields and methods. The fields are effectively declarations of data while the methods define routines that act on the data. Object types allow four types of routines: procedures, functions, and also constructors and destructors; the latter two are allowed only within objects.

See also **Object Syntax**.

Inheritance

One object type can extend another object type by adding or replacing fields and methods. In this case the new object is said to be a descendant object; the older object is said to be an ancestor object. The process of an extension is called its inheritance. The descendant object type may have its own descendants; these are also viewed as the descendants of the original ancestor object. The domain of an object, is the object together with all of its descendants.

Object Syntax

The syntax of an object type declaration is

```
object [heritage]
  Component list
  [ private Component list ]
end
```

where *Component list* is defined as:

```
[Fieldlist]
  [Method list]
Heritage:
  (object type identifier)
Fieldlist:
  [ field entry [; Field list ] ]
Field entry:
  identifier list : type
MethodList:
  [ method entry [; MethodList ] ]
Method entry:
  [ method heading [; virtual ] ]
```

Restrictions On Object Description

object types can be declared anywhere a type identifier is allowed by Pascal's syntax. **object** types can be declared within procedures, functions, or other methods if the declaration is not ambiguous.

Like records, object types cannot include *File* components or records or objects that include 'file' components.

OOP Scopes

Component identifiers are visible in all the methods throughout the domain of the object, including the procedures, functions, destructors and constructors that implement the methods of the object type and its descendants. However, the scopes of the fields and methods declared in the **private** section of the object type declaration are restricted to the unit that contains the definition of the object type. **private** fields and methods are inaccessible from other units. Private fields and methods can, however, be accessed from other object types declared in the same unit. Below are examples of several objects:

```
type Point =
  record
    X,Y: Longint;
  end

type Circle =
  object
    Center: Point;
    Radius: Longist;
    procedure Show;
    procedure Hide;
  end;

type Ellipse =
  object (Circle)
    Radius2: Longint;
```

```
    Angle: Real;
    procedure Show;
    procedure Rotate(NewAngle: Real);
  end;
```

Here, the object type *Ellipse* 'inherits the *Center* and *Radius* fields from *Circle*. It also adds a new *Radius2* and *Angle*. Furthermore, it uses the method *Hide* inherited from *Circle*; it overrides the method *Show* and adds a new method, *Rotate*. The declaration of an object file includes just the headers of the methods. The methods themselves should appear somewhere within the current scope. In this way, method declarations are similar to forwarded routines. When specified, methods names are qualified with object names. For example

```
procedure Circle.Draw;
begin
  Graph.Circle(X,Y,Radius);
end;
```

Note that within the method declaration, the fields of the object are visible to the compiler.

Public and Private declarations

Public and **private** are standard directives in the Object Pascal language. Treat them as if they were reserved words. For readability, it is best to organize an object declaration by visibility, placing all the **private** members together, followed by all the **protected** members, and so on. This way each visibility reserved word appears at most once and marks the beginning of a new section of the declaration. So a typical object declaration should look like this:

```
type
  TObject = object
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

The scope of component identifiers declared in **private** component sections is restricted to the module that contains the object type declaration. Keep in mind that:

- Inside the module, **private** component identifiers act like normal **public** component identifiers.
- Outside the module, **private** component identifiers are unknown and inaccessible.

Use the **public** part to

- Declare data fields you want methods in objects in other units to access
- Declare methods you want objects in other units to access

Declarations in the **private** part are restricted in their access. If you declare fields or methods to be **private**, they are unknown and inaccessible outside the unit the object is defined in. Use the private part to

- Declare data fields you want only methods in the current unit to access
- Declare methods you want only objects defined in the current unit to access

Virtual Methods

Methods can be either static or virtual. Calls to static methods are resolved at compilation. Calls to virtual methods are resolved at run time with delayed or late binding. By default the methods are static; virtual methods contain a special keyword **virtual** as part of their declaration. Static methods can be overridden without restrictions. However, virtual method override must be done by a method that uses exactly the same syntax, e.g. has the same number and types of the arguments. Objects that contain virtual methods require building a special jump table, called the Virtual Method Table (VMT). The VMT is created during the initialization of the object through a constructor call.

Constructors

Constructors initialize (instantiate) objects by creating and filling their VMT. Any object that uses virtual methods must be first initialized.

```
type Circle =
  object
    Center: Point;
    Radius: Longint;
    constructor Init(Z:Point; R:Longint);
    procedure   Show; virtual;
    procedure   Hide;
    destructor  Kill;
  end;
var
  C: Circle;
  P: Point;
```

The following code will instantiate, display, hide, and display the circle *C*:

```
P.X:=20;
P.X:=40;
C.Init(P,100);
C.Show;
While Not KeyPressed Do;
C.Hide;
C.Kill;
```

where:

```
constructor Circle.Init(Z:Point; R:Longint);
begin
  Center:=P;
  Radius:=R;
end;
```

Besides initializing the fields of the circle *C*; *C.Init* also creates a VMT table. This table is essential for calling a virtual method, such as *Show*.

Without the *C.Init* call, the example above will fail (probably cause a run-time exception or halt the system). However, when the example is compiled with the range-check `{$R+}` switch on, TMT Pascal will automatically detect calls from a non-instantiated method and produce a run-time error.

See also **Destructors**

Fail procedure

Called from within a constructor, *Fail* causes the constructor to de-allocate a dynamic object it has just allocated.

Declaration:

```
procedure Fail;
```

Remarks:

Fail must be called only if one of the constructor's operations fails.

Using New Procedure (OOP)

In most cases, instantiating of an object is combined with allocation of memory for the object:

```
var C: ^Circle;
begin
  New(C);
  C.Init(P, R);
...
```

The extended syntax of the New procedure allows one to combine the operation:

```
var C: ^Circle;
begin
  New(C.Init(P, R));
...
```

Note that constructors cannot be virtual methods, since virtual methods cannot be called before a constructor initializes the VMT.

See also **New**

Desctructors

Destructors are used to clean up after an object is no longer needed. Unlike constructors, destructors can be virtual. Destruction of an object is often combined with deallocation of its memory with the **Dispose** procedure:

```
var C: ^Circle;
begin
  New(C);
  C.Init(P,R);
  ...
  C.Kill;
  Dispose(C);
end;
```

If a constructor fails to perform initialization (often because of its inability to allocate memory) for the structures affiliated with the object, it can execute a special system function *Fail*. *Fail* signals TMT Pascal to reverse all allocation of the object that might have occurred and return *nil* as the value of the object's pointer. *Fail* can be called only within constructors.

See also **Constructors**

Inherited reserved word

Inherited can be used to denote the ancestor of the enclosing method's object type. **inherited** cannot be used within methods of an object type that has no ancestor.

Self argument

Methods have an additional implicit argument, called *Self*, which is automatically supplied by the compiler. *Self* contains the instance of the object for which the method was called. *Self*, as well, as all of its fields are automatically added to the method's symbol table.

1.12 Open Arrays

TMT Pascal allows one to use a multidimensional open array as a parameter in procedures and functions. The open array parameter has the following format description:

```
array [dim] of type,
```

where *dim* is a positive integer constant, defining the number of dimensions, and type is the type of the array elements. To determine the upper bounds of the array, use the high (array) function. It returns a vector of Longints (**array** [0..dim-1] **of** Longint) containing the upper bounds. The lower bounds are always set to 0. The vector of the lower bounds can be obtained with a **Low** function.

Example:

```
procedure print_vector (v: array(1) of Double);
var
  i: integer;
begin
  for i := 0 to high(v)[0] do Write(v[i]:10:6, ' ');
  Writeln;
end;
procedure print_matrix(m: array(2) of double);
var
  i: integer;
begin
  for i := 0 to high(m)[0] do print_vector(m[i]);
  Writeln;
end;
const a: array[1..3, 1..3] of Double =
((1,0,2),(2,1,0),(1,2,1));
begin
  print_matrix(a);
end.
```

1.13 User Defined Operators

TMT Pascal allows redefining of the standard operators on predefined types and overloading of these operators for new types. For this, it uses the construction

```
overload
```

The syntax is:

```
overload op_sign = qualified procedure identifier;
```

Where the *op_sign* is one of the standard operator symbols:

```
+ - / * = <> < > <= >=
and or xor shl shr mod div in not
+:=   -:=   *:=   /:=
```

When a re-defined operator is used, TMT Pascal uses the last definition that could be applied toward operands of given types. For example, this fragment:

```
function add2_rr (a, b:    Real): Real;
  Result := (a + b) * 2;
function add2_ii (a, b: Integer): Integer;
  Result := (a + b) * 2;

overload + = add_rr;
overload + = add_ii;
```

redefines the "+" operator. Notice that the order of overload's is important. The reverse order

```
overload + = add_ii;
overload + = add_rr;
```

will cause *add_rr* to be used always since integers can always be cast into reals.

In the SOURCES subdirectory you can find the source of the COMP module which realizes the complex numbers and defines the operators on them.

Remarks:

- The operators +:=, -:=, *:= and /:= have the lowest precedence (lower, than the comparison operators) and are right-associative.
- The operators "+:=" and "-:=" are predefined for all integer and real types.
- The operators "*:=" and "/:=" are predefined for all real types, with the obvious meaning.

1.14 Built-in Assembler

TMT Pascal allows mixing assembly language code with Pascal using two distinct methods:

Using external assembly files, compiled with a suitable 32-bit assembler. These can be linked in with the **{$L}** Pascal directive.

Using built-in assembly code, which can be placed inside Pascal source files.

This chapter describes the built-in assembler (BASM).

Since the program created by the TMT Pascal is executed in the flat model, the far call and jump commands as well as the @Code and @Data symbols are not implemented.

Asm Statement

The built-in assembler is invoked with the **asm** statement. The syntax of the **asm** statement is

```
asm
  [AssemblerStatement(s)]
end;
```

The **asm** statement may appear anywhere where a Pascal statement allowed.

```
asm
  MOV  Al, Value
  MOV  DX, ThePort
  OUT  DX, AL
end;
```

Assembler Procedure

The built-in assembler can also be used to write entire procedures in assembler language. Such procedures should have the **assembler** keyword appended after a procedure header.

```
function MultBy9(X: Longint):Longint;
assembler;
  asm
    MOV  EAX,[X]
    LEA  EAX,[EAX*8+EAX]
end;
```

The function above used the i80386 index scaling feature to implement very fast multiplication by 9.

Assembler procedures differ from the standard Pascal procedures in the following ways:

No Return variable
There is no return variable. You must return the function results in an appropriate register. More precisely,

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real values are returned in DX:BX:AX.
- Floating point (8087) values are returned in ST(0).
- Pointers are returned in EAX.
- Strings are returned in a temporary location pointed by the *@Result* symbol.

Structured variables
Structured arguments (i.e. strings, objects, records) are not copied into the local variables. They should be treated as **var** parameters.

Stack Frame
Assembler procedures have no stack frames if they have no arguments and no local symbols. Generally, the stack frame supplied by the built-in assembler is

```
PUSH   EBP          // Appears if locals + params >0
MOV    EBP,ESP      // Appears if locals + params >0
SUB    ESP, locals  // Appears if locals + params >0
...
LEAVE               // Appears if locals + params >0
RETN   params       // Always appears
```

Here *Locals* is the total size of local parameters, *Params* is the total size of procedure parameters.

Register Preservations
Assembler code should preserve the following registers: DS, CS, SS, ES, EBP, and ESP. All other registers can be destroyed. Notice the inclusion of the ES register. TMT Pascal always assumes that ES is equal to DS.

Do not change segment, page, and interrupt tables, as well as the control, debug and test registers, unless you are thoroughly familiar with 386 protected mode architecture. Privileged instructions like LGDT and LIDT are supported by built-in assembler. However, avoid using them unless you know exactly what you are doing.

Code Procedure

Besides the assembler-routine you can use the code-routine. It has the following differences: the compiler doesn't emit the frame command on enter and return from the routine (including the ret command), and the local parameters are based on ESP at the moment of entry.

Example:

```
function hi (n: word); code;
  asm
    mov al, byte ptr [n+1]
    ret
end;
```

Command Syntax

The general syntax of an assembler statement is

```
[label:]
 [prefixes]
 [[opcode [operand1 [,operand2 [,operand3]]]]
```

Here:

label: is an optional label definition;
prefixes are instruction prefixes;
opcode is a instruction mnemonic or directive;
operand is an operand expression.

Assembler Labels

The built-in assembler allows two types of labels:

- Global labels are declared inside a Pascal program within label declarations. Global labels are identical to Pascal labels.
- Local labels are not declared. They must start with the @ symbol and contain letters, digits, or underscore characters. Local labels are visible only within the current **asm** statement.

Labels can be used with any assembler statements. More than one label can be used, if needed. Labels are always optional.

Assembler Prefixes

Prefixes are modifiers for the following instruction. TMT Pascal allows the following prefix mnemonics:

```
* SEGCS           Override the operand's segment with CS:
* SEGDS           Override the operand's segment with DS:
* SEGES           Override the operand's segment with ES:
* SEGFS           Override the operand's segment with FS:
* SEGGS           Override the operand's segment with GS:
* SEGSS           Override the operand's segment with SS:
* LOCK            Lock the bus
* REP             Repeat the instruction
* REPE  and REPZ  Repeat while equal
* REPNE and REPNZ Repeat while not equal
```

Assembler Opcodes

The opcodes are either instruction mnemonics or assembly directives. The list of supported instruction opcodes is given below. The only assembly directives that are allowed in TMT Pascal are DB, DW, and DD.

Example:

```
asm
  DB  'a','b','c'
  DB  'This code was copyrighted by GnueWare'
  DW  1,2,4,8,16,$20,40h
  DD  Offset HeapLo
end;
```

The DB, DW, and DD directives allow a variable number of arguments, separated by commas. The other commonly used assembly directives can be emulated with Pascal statements. For instance, the EQU directive is emulated with **const**, while STRUCTs can be defined with the **type record** declaration.

Assembler Registers

The following registers can appear in built-in assembler:

```
8-bit:      AL BL CL DL AH BH CH DH
16-bit:     AX BX CX DX SI DI BP SP
32-bit:     EAX EBX ECX EDX ESI EDI EBP ESP
Segment:    CS DS ES SS FS GS
8087:       ST
Control:    CRn
Debug:      DRn
Test:       TRn
```

The segment register can be used for segment overrides. 32-bit registers can be used for indexing following the standard 80386 conventions. 16-bit registers should never be used for addressing, unless your entire program does not exceed 64K. Even then, addressing with 16-bit registers is inefficient. Generally, an address is formed as

```
Base + Index * Scale + Displacement
```

where *Base* is any of the 32-bit registers, *Index* is any 32-bit register but not ESP, and *Scale* should be 1,2,4, or 8. Finally, the *Displacement* is a 32-bit integer quantity.

Here are some valid and invalid indexing modes:

```
  [EAX+EBX]; ok
  [EAX+EAX]; ok, EAX is both base and index.
      [ESP]; ok, ESP is index, no base.
    [EDX*2]; ok, use this to index a global array of words.
[EAX*4+EBP]; ok, use this to index a local array of longints.
       [SI]; ok, but is likely to lead to hard-to-find bugs.
   [ESI+BX]; illegal, mix of 16- and 32- bit registers.
     [SI*4]; illegal, 16-bit registers cannot be scaled.
    [ESP*4]; illegal, ESP cannot be an index.
```

Please consult Intel™ 80386/80486 Programmer's reference for more details.

Assembler Opcode Mnemonics

This section lists valid opcode mnemonics. Please consult an 80386 reference book for additional details. The list uses the following abbreviations:

```
* acc   - accumulator register (AL, AX, EAX)
* brm   - byte register or memory operand
* cdt   - control, debug or test register
* imm   - byte
* label - offset in code
* mem   - memory operand
* none  - no operands
* reg   - register
* rm    - register or memory operand
* seg   - segment register
* st    - coprocessor top of stack register
* st(i) - coprocessor register
```

Opcode:	Possible arguments
AAA:	none
AAD:	none
AAM:	none
AAS:	none
ADC:	rm,reg | reg,rm | rm,imm
ADD:	rm,reg | reg,rm | rm,imm
AND:	rm,reg | reg,rm | rm,imm
ARPL:	rm,reg
BOUND:	reg,mem
BSF:	reg,rm
BSR:	reg,rm
BTC:	rm,reg | rm,imm
BTR:	rm,reg | rm,imm
BTS:	rm,reg |rm,imm
BT:	rm,reg | rm,imm
CALL:	label | rm
CBW:	none
CDQ:	none
CLC:	none
CLD:	none
CLI:	none
CLTS:	none
CMC:	none
CMP:	rm,reg | reg,rm | rm,imm
CMPSB:	none
CMPSD:	none
CMPSW:	none
CPUID:	none
CWD:	none
CWDE:	none
DAA:	none
DAS:	none
DEC:	rm
DIV:	rm
ENTER:	imm,imm
F2XM1:	none
FABS:	none
FADD:	none | st,st(i) | st(i),st | mem
FADDP:	st(i),st
FBLD:	mem

```
  FBSTP:      mem
   FCHS:      none
  FCLEX:      none
   FCOM:      none | st(i) | mem
  FCOMP:      none | st(i) | mem
 FCOMPP:      none
FDECSTP:      none
  FDISI:      none
   FDIV:      none | st,st(i) | st(i),st | mem
  FDIVP:      st(i),st
  FDIVR:      st(i),st
 FDIVRP:      st(i),st
   FENI:      none
  FFREE:      st(i)
  FIADD:      mem
 FICOMP:      mem
  FICOM:      mem
 FIDIVR:      mem
  FIDIV:      mem
   FILD:      mem
  FIMUL:      mem
  FIMUL:      mem
FINCSTP:      mem
  FINIT:      none
   FIST:      mem
  FISTP:      mem
  FISUB:      mem
 FISUBR:      mem
    FLD:      st(i) | mem
   FLD1:      none
  FLDCW:      mem
 FLDENV:      none
 FLDL2E:      none
 FLDL2T:      none
 FLDLG2:      none
 FLDLN2:      none
  FLDPI:      none
   FLDZ:      none
   FMUL:      none | st,st(i) | st(i),st | mem
  FMULP:      none | st(i),st
 FNCLEX:      none
 FNDISI:      none
  FNENI:      none
 FNINIT:      none
   FNOP:      none
 FNSAVE:      none
 FNSTCW:      none
FNSTENV:      none
 FNSTSW:      none
 FPATAN:      none
  FPREM:      none
 FPREM1:      none
  FPTAN:      none
FRNDINT:      none
 FRSTOR:      mem
  FSAVE:      mem
 FSCALE:      none
 FSETPM:      none
  FSQRT:      none
```

```
    FST:      st(i) | mem
   FSTP:      st(i) | mem
  FSTCW:      mem
 FSTENV:      mem
  FSTSW:      mem | AX
   FSUB:      none | st,st(i) | st(i),st | mem
  FSUBP:      none | st(i)st
  FSUBR:      none | st,st(i) | st(i),st | mem
 FSUBRP:      none | st(i),st
   FTST:      none
  FWAIT:      none
   FXAM:      none
   FXCH:      none
FXTRACT:      none
FYL2XP1:      none
  FYL2X:      none
    HLT:      none
   IDIV:      rm
   IMUL:      rm | reg,imm | reg,rm,imm
     IN:      acc,imm | acc,DX
    INC:      rm
   INSB:      none
   INSD:      none
   INSW:      none
    INT:      imm
   INTO:      none
  IRETD:      none
   IRET:      none
     JA:      label
    JAE:      label
     JB:      label
    JBE:      label
     JC:      label
   JCXZ:      label
     JE:      label
  JECXZ:      label
     JG:      label
    JGE:      label
     JL:      label
    JLE:      label
    JMP:      label | rm
    JNA:      label
   JNAE:      label
    JNB:      label
   JNBE:      label
    JNC:      label
    JNE:      label
    JNG:      label
   JNGE:      label
    JNL:      label
   JNLE:      label
    JNO:      label
    JNP:      label
    JNS:      label
    JNZ:      label
     JO:      label
     JP:      label
    JPE:      label
    JPO:      label
     JS:      label
```

```
    JZ:      label
  LAHF:      none
   LAR:      reg,rm
   LDS:      reg,mem
 LEAVE:      none
   LEA:      reg,mem
   LES:      reg,mem
   LFS:      reg,mem
  LGDT:      mem
   LGS:      reg,mem
  LIDT:      mem
  LLDT:      rm
  LMSW:      rm
 LODSB:      none
 LODSD:      none
 LODSW:      rm
  LOOP:      label
LOOP16:      label
LOOP32:      label
 LOOPE:      label
LOOPNE:      label
LOOPNZ:      label
 LOOPZ:      label
   LSL:      reg,rm
   LSS:      reg,mem
   LTR:      reg,mem
   MOV:     reg,rm|rm,reg|rm,imm|rm,seg|seg,rm|reg,cdt|cdt,reg
 MOVSB:      none
 MOVSD:      none
 MOVSW:      none
 MOVSX:      reg,rm
 MOVZX:      reg,rm
   MUL:      rm
   NEG:      rm
   NOP:      none
   NOT:      rm
    OR:      none
   OUT:      imm,acc | DX,acc
 OUTSB:      none
 OUTSD:      none
 OUTSW:      none
   POP:      rm
  POPA:      none
 POPAD:      none
  POPF:      none
 POPFD:      none
  PUSH:      rm
 PUSHA:      none
PUSHAD:      none
 PUSHF:      none
PUSHFD:      none
   RCL:      rm,1 | rm,CL | rm,imm
   RCR:      rm,1 | rm,CL | rm,imm
   RET:      none,imm
  RETF:      none,imm
  RETN:      none,imm
   ROL:      rm,1 | rm,CL | rm,imm
   ROR:      rm,1 | rm,CL | rm,imm
  SAHF:      none
```

```
   SAL:      rm,1 | rm,CL | rm,imm
   SAR:      rm,1 | rm,CL | rm,imm
   SBB:      rm,reg | reg,rm | rm,imm
 SCASB:      none
 SCASD:      none
 SCASW:      none
 SEGCS:      none
 SEGDS:      none
 SEGES:      none
 SEGSS:      none
 SEGFS:      none
 SEGGS:      none
  SETA:      brm
 SETAE:      brm
  SETB:      brm
 SETBE:      brm
  SETC:      brm
  SETE:      brm
 SETGE:      brm
  SETG:      brm
  SETL:      brm
 SETLE:      brm
 SETNA:      brm
SETNAE:      brm
SETNBE:      brm
 SETNB:      brm
 SETNC:      brm
 SETNE:      brm
SETNGE:      brm
 SETNG:      brm
SETNLE:      brm
 SETNL:      brm
 SETNO:      brm
 SETNP:      brm
 SETNS:      brm
 SETNZ:      brm
  SETO:      brm
 SETPE:      brm
 SETPO:      brm
  SETP:      brm
  SETS:      brm
  SETZ:      brm
  SGDT:      mem
  SHLD:      rm,reg,imm
   SHL:      rm,1 | rm,CL | rm,imm
  SHRD:      rm,reg,imm
   SHR:      rm,1 | rm,CL | rm,imm
  SIDT:      mem
  SLDT:      rm
  SMSW:      rm
   STC:      none
   STD:      none
   STI:      none
 STOSB:      none
 STOSD:      none
 STOSW:      none
   STR:      rm
   SUB:      rm,reg | reg,rm | rm,imm
  TEST:      rm,reg | reg,rm | rm,imm
  VERR:      rm
```

```
VERW:      rm
WAIT:      none
XCHG:      reg,rm | rm,reg
XLAT:      none
 XOR:      rm,reg | reg,rm | rm,imm
```

Assembler Operand Expressions

Operand expressions are built from operands and operators. Operands are constants, registers, labels, and memory locations. Operators combine operands and alter their attributes. Each instruction allows only certain combinations of operands.

Operand expressions can be classified into three classes:

- Immediate operands or constants.
- Registers.
- Memory and label operands.

Immediate operands

```
PUSH  10
MOV   AX, 10
MOV   AX, offset Start
```

Here *10*, and *Star*' are immediate operands. The values of *A* and *10* can be determined immediately; the value of *offset Start* is determined during linking.

Registers

TMT Pascal built-in assembler allows use of any 8, 16, or 32-bit 80386 registers.

```
XOR  AH, AL
LSL  EAX,EAX
MOV  AX, FX
```

Memory and label operands

Memory operands refer to the data stored in memory locations. Usually, one uses square brackets [..] or the *type ptr* operator to ensure that the argument is treated as a memory location. Label operands refer to locations in code.

```
MOV      EBX,[0]
POP      Word Ptr [88]
POP      Word Ptr 88  // same as above
JMP      @exit
LOOP16   @loop
```

Memory and immediate operands can be either absolute or relocatable. An operand is absolute if its value or offset is entirely known during compilation. An operand is relocatable if its offset will become known only during linking.

Assembler Operands

TMT Pascal allows the following operands: Numeric Constants, Strings, Registers, Pascal Symbols and Special Assembler Symbols.

Numeric Constants

Numeric constants are 32-bit integers, e.g. integers in the range -2147483648..4294967295. Numeric constants can be entered as decimal numbers, binary numbers (using the 'B' suffix),

octal numbers (using the 'O' or 'Q' suffix) or hexadecimal numbers (using the 'H' suffix or '$' prefix). Note that the hexadecimal numbers must start with a digit.

Strings

Strings are enclosed in either single or double quotes. A repeated quote of the same type as the surrounding quotes is treated as one character. String constants of arbitrary length may occur only in the 'DB' directive. In all other cases, the string must not exceed four characters and its value is converted into an integer number.

Registers

The use of registers was described above.

Pascal Symbols

With built-in assembler you can access the majority of Pascal symbols. These include labels, constants, variables, types, and procedures. The values, classes and types of Pascal symbols are summarized in the table below:

Symbol	Value	Class	Type
label	its address	Memory	NEAR
constant	its value	Immediate	0
type	0	Memory	sizeof(type)
field	its offset	Memory	sizeof(type)
variable	its address	Memory	sizeof(type)
procedure	its address	Memory	NEAR
function	its address	Memory	NEAR
unit	its address	Immediate	0

Special Assembler Symbols

Built-in assembler supports five special symbols: @CODE, @DATA, @RESULT, @PARAMS and @LOCALS. The @CODE and @DATA are not really useful in a flat model. They always return 0Ch and 14h, which are the standard segment selectors for the code and data segments. The @RESULT symbol points to the pseudo-variable that contains the function return, @PARAMS and @LOCALS return the size of the parameter and local areas on stack.

Assembler Operators

TMT Pascal built-in assembler syntax allows the a number of operators, listed below:

&	Identifier override operator. The following identifier is considered to be a user defined symbol, even if its spelling is identical to an assembler reserved word.
()	Parenthesis. Expressions within parenthesis are evaluated first.
[]	Memory reference. The expression within brackets is evaluated first. This expression should be a valid 386 address. The resulting expression is always treated as a memory reference.
HIGH, LOW	High and Low byte selection. These operators return the high and low 8 bits of the word-size expression that follows.
+, -	Unary plus and minus operators. The expression should be an absolute immediate.
SMALL, LARGE	Forces the built-in assembler to treat the following operands as a 16- or 32- bit quantity.
OFFSET	Returns the 32-bit offset of the expression that follows.
SEG	Returns the segment part of the operand.
TYPE	Returns the type of the operand. The type of a NEAR symbol is -1, of a FAR symbol is -2. The type of a memory

operand is its size. The type of an immediate symbol is 0.

PTR Typecasts the following expression into the type symbol that precedes the PRT operator. Valid typecasts are BYTE PRT, WORD PRT, DWORD PRT, FWORD PRT, TBYTE PRT, QWORD PRT, NEAR PRT, and FAR PRT.

***** Multiplication. Both arguments must be absolute immediate quantities, or one of the arguments should be an index register and the other a scale factor (1,2,4, or 8).

/ Division. Both arguments must be absolute immediate quantities.

MOD Integer remainder after division. Both arguments must be absolute immediate quantities.

SHL, SHR Left and right shifts. Both arguments must be absolute immediate quantities.

+, - Addition and Subtraction. At most one argument can be a relocatable value and it cannot be the argument that is being subtracted. The other argument must be an absolute immediate quantity.

NOT Binary complement. The argument must be an absolute immediate quantity.

AND Bitwise AND. The arguments must be absolute immediate quantities.

OR Bitwise OR. The arguments must be absolute immediate quantities.

XOR Bitwise exclusive OR. The arguments must be absolute immediate quantities.

Assembler Operator Precedence

The precedence of these operators is shown in the following table (from the highest precedence to the lowest):

Operator	Comments
&	Identifier override
() []	Sub-expressions, memory reference, structure member
HIGH LOW	High and low bytes selectors
LARGE SMALL	32- and 16- bit operation overrides
+ -	Unary operators
:	Segment override
OFFSET SEG TYPE PTR */MOD SHL SHR	
+ -	Binary operators
NOT AND OR XOR	Bitwise operators

Differences between 16- and 32-bit code

While this manual does not really teach 32-bit assembly programming, we will list here several considerations important in 32-bit assembler programming. These considerations may be helpful to a 16-bit programmer entering the 32-bit arena.

Avoid using 16-bit registers for indexing
The built-in assembler will correctly assemble instructions like

```
MOV       BX,offset table
MOV       AX,table[BX]
JMP       table[BX]
```

However, these instructions will not work correctly if the size of your program exceeds 64K and the 'table' variable is placed after the 64K limit. This is because 16-bit addresses span only the 64K of the segment. The last example above is the most dangerous; it is likely to crash the system.

Jump tables

Jump tables should be built as tables of 32-bit addresses, not 16-bit addresses.

Longint (32-bit) Arithmetic

Try to use longint arithmetic as much as possible. 16-bit instructions often take more space than corresponding 32-bit instructions. In

```
XOR     AX,    AX
MOV     data1, AX
MOV     data2, AX
```

it would be better to replace the first instruction with

```
XOR     EAX, EAX
```

which is one byte shorter. Furthermore, if *data1* and *data2* can be changed into longints, you may save a lot more space (and time) both in the assembler and the Pascal sections of the program.

ECX vs CX

Loop and repeat instructions in 32-bit mode use the ECX register rather than CX. The following program segment is likely to cause problems:

```
MOV    CX,  size
MOV    ESI, source
MOV    EDI, dest
REP    MOVSB
```

Also notice that the source and destination registers are ESI and EDI, rather than SI and DI.

POPAD/PUSHAD

Use POPAD and PUSHAD instead of POPA and PUSHA. The latter instructions generate only 16-bit pushes.

POPFD/PUSHFD

Use POPFD and PUSHFD instead of POPF and PUSHF. The latter instructions generate only 16-bit pushes.

IRETD

Use 'IRETD' instead of 'IRET'. The latter instruction pops 16-bit registers.

String instructions

When doing string operations, it is better to use double word instructions instead of byte or word. Use MOVSD instead of MOVSW or MOVSB.

JECXZ vs JCXZ

Distinguish between the JCXZ and JECXZ instructions. The former tests the CX register, while the latter tests ECX. Use of JCXZ instead of JECXZ may lead to hard-to-find bugs. Similarly, LOOP tests ECX, while LOOP16 tests CX.

Function results

Remember to return 32-bit results in EAX, not DX:AX.

ES: preservations

Do not change the ES register. TMT Pascal depends on ES = DS.

Immediate PUSH

TMT Pascal assumes that an immediate push instruction like

```
  PUSH    Small 0
  PUSH    Small offset data
```

Furthermore notice that like TASM and unlike the PharLap assembler, TMT Pascal will treat

```
  PUSH    Word Ptr 0
```

as if it were

```
  PUSH   Word Ptr [0]
```

Var Parameters

Similar to 16-bit mode, **Var** parameters are 32-bit pointers. However, in TMT Pascal, pointers are just 32-bit offsets within the data segment. Therefore, **Var** parameters are retrieved with a 'MOV' instruction, not with an LES or an LDS.

Local Symbols

Local Symbols and Parameters are addressed via the EBP register. For example, in

```
var local: Longint;
asm
  MOV   EAX, local
end;
```

the last line assembles into

```
  MOV   EAX, [EBP-4]
```

1.15 Standard Units

TMT Pascal comes with a set of standard units (see UNITS.PDF for more info). It is also possible to create units. For example, in writing a large program it might become desirable to group display routines or user input routines. This allows for greater organization while programming. For more information on creating your own units see the Programs and Units chapter.

Chapter 2

Win32 Programming

2.1 Writting Win32 GUI Applications

TMT Pascal produces native Win32® GUI applications. This chapter is based on the Microsoft® Win32® Programmer's Reference and describes particularities of GUI application development using the TMT Pascal Multi-target. The TMT Pascal compiler comes with a set of units which define function and procedure headers for the Windows API.

For more information refer to **Microsoft Win32 Programmer's Reference** and **Microsoft Multimedia Programmer's Reference**. Also, you will find sources of all Win32 API interface units in the **\TMTPL\SOURCE\WIN32** subdirectory.

Every graphical Win32-based application creates at least one window (called the main window) that serves as the main window for the application. This window serves as the primary interface between the user and the application. Most applications also create other windows, either directly or indirectly, to perform tasks related to the main window. Each window plays a part in displaying output and receiving input from the user.

At the start of an application, the system associates a taskbar button with the application. The taskbar button contains the program icon and the title. When the application is active, its taskbar button is displayed in the pushed state.

2.2 Structure of Window Procedure

A window procedure is a function that has four parameters and returns a 32-bit signed value (Longint). The parameters consist of a window handle, a UINT message identifier, and two message parameters declared with the WParam and LParam data types. For more information, see **WindowProc**.

Message parameters often contain information in both their low-order and high-order words. The Microsoft® Win32® application programming interface (API) includes several macros an application can use to extract information from the message parameters. The LOWORD function, for example, extracts the low-order word (bits 0 through 15) from a message parameter. Other functions include HIWORD, LOBYTE, and HIBYTE.

The interpretation of the return value depends on the particular message. Consult the description of each message to determine the appropriate return value.

Because it is possible to call a window procedure recursively, it is important to minimize the number of local variables that it uses. When processing individual messages, an application should call functions outside the window procedure to avoid excessive use of local variables, possibly causing the stack to overflow during deep recursion.

2.3 Designing a Window Procedure

The following example shows the structure of a typical window procedure. The window procedure uses the message argument in a **CASE** statement to process. For messages that it does not process, the window procedure calls the DefWindowProc function.

```
function MainWndProc conv arg_stdcall (
    _hwnd: HWND,        // handle of window
    _uMsg: UINT,        // message identifier
    _wParam: WPARAM,    // first message parameter
    _lParam: LPARAM     // second message parameter
  ): LRESULT;
begin
    case _uMsg of
        WM_CREATE:
          begin
            // Initialize the window.
            Result := 0;
          end;

        WM_PAINT:
          begin
            // Paint the window's client area.
            Result := 0;
          end;

        WM_SIZE:
          begin
            // Set the size and position of the window.
            Result := 0;
          end;

        WM_DESTROY:
          begin
            // Clean up window-specific data objects.
            Result := 0;
          end;
        //
        // Process other messages.
        /

        else
            Result := DefWindowProc(_hwnd, _uMsg, _wParam,
_lParam);
    end;
end;
```

2.4 Associating a Window Procedure with a Window Class

One associates a window procedure with a window class when registering the class. You must fill a **TWndClass** structure with information about the class, and the lpfnWndProc member must specify the address of the window procedure. To register the class, pass the address of **TWndClass** structure to the **RegisterClass** function. Once the window class is registered, the window procedure is automatically associated with each new window created with that class.

The following example shows how to associate the window procedure in the previous example with a window class:

```
var
  wc: TWndClass;
begin
    // Register the main window class.
  with wc do begin
    style := CS_HREDRAW or CS_VREDRAW;
    lpfnWndProc := @MainWndProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackground := GetStockObject(WHITE_BRUSH);
    lpszMenuName :=  'MainMenu';
    lpszClassName := 'MainWindowClass';
  end;
    if not RegisterClass(wc) then MyError;

    //
    // Process other messages.
    //
end.
```

2.5 Example of a Win32 GUI Application

```
program Hellow;

{$ifndef __WIN32__}
{$define INVALID_TARGET}
{$endif}
{$ifndef __GUI__}
{$define INVALID_TARGET}
{$endif}
{$ifdef INVALID_TARGET}
  This program must be compiled for Win32 GUI target only
{$endif}

uses Windows, MMSystem, Messages;

function MyWndProc conv arg_stdcall (Window: HWND; Mess: UINT;
Wp: WParam; Lp: LParam): LRESULT;
begin
  case Mess of
    WM_PAINT:       begin
                      declare
                      var
                        DC: hDC;
                        ps: TPaintStruct;
                      begin
                        DC := BeginPaint(Window, ps);
                        TextOut(DC, 0, 0, 'Hello World!', 12);
                        EndPaint(Window, ps);
                        Result := 0;
                      end;
                    end;
```

```
    WM_DESTROY:     begin
                      PostQuitMessage(0);
                      Result := 0;
                    end;
    WM_LBUTTONDOWN: begin
                      MessageBox(Window, 'This is my message!',
                        'My message box', MB_OK);
                      Result := 0;
                    end;
   else
                  Result := DefWindowProc(Window, Mess, Wp,
Lp);
  end;
end;

var
  wc : TWndClass;
  wnd: HWnd;
  Msg: TMsg;
begin
  FillChar(wc, SizeOf(wc), 0);
  with wc do begin
    style:=CS_HREDRAW + CS_VREDRAW;
    lpfnWndProc := @MyWndProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := System.hInstance;
    hIcon := LoadIcon(THandle(NIL), IDI_APPLICATION);
    hCursor := LoadCursor(THandle(NIL), IDC_ARROW);
    hbrBackGround := COLOR_WINDOW+1;
    lpszMenuName := nil;
    lpszClassName := 'HelloWorld';
  end;
  if RegisterClass(wc) = 0 then
  begin
    Exit;
  end;

  wnd := CreateWindow(wc.lpszClassName, 'GUI Application Demo',
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, 0, 0,
HInstance, NIL);

  ShowWindow(wnd, SW_RESTORE);
  UpdateWindow(wnd);

  while GetMessage(Msg,0,0,0) do
  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;

end.
```

2.6 Writting Win32 Control Panel Applications

Even though Windows provides a number of standard **Control Panel applications (CPL)**, one can create additional applications with TMT Pascal to let users examine and modify the settings and operational modes of specific hardware and software.

2.7 Application Responsibilities and Operation

The primary responsibility of any Control Panel application is to display a dialog box and to carry out any tasks specified by the user. Despite this responsibility, Control Panel applications do not provide menus or other direct means for users to access their dialog boxes. Instead, these applications operate under the control of another application and display their dialog boxes only when requested by the controlling application.

Control Panel applications are usually controlled by a Windows system utility specifically designed to give users access to these applications. However, any application can load and manage Control Panel applications, as long as the controlling application sends messages and processes return values in the way that the Control Panel applications expect.

Most Control Panel applications display and manage a single dialog box, giving the user control of the settings and operational modes of a single system component. However, any given Control Panel application can provide any number of dialog boxes to control any number of system components. (These individual dialog boxes are sometimes called applets.) To distinguish between dialog boxes, a Control Panel application typically supplies the controlling application with a unique icon for each dialog box. The controlling application displays these icons and the user can choose a dialog box by choosing the corresponding icon.

2.8 Application Entry-Point Function

Every Control Panel application must export the standard entry-point function, **CPlApplet**. This function receives requests, in the form of Control Panel (CPL) messages, and carries out the requested work, such as initializing the application, displaying and managing the dialog box(es), and closing the application.

When the controlling application first loads the Control Panel application, it retrieves the address of the **CPlApplet** function and subsequently uses the address to call the function and pass it messages. The controlling application may send the following messages:

CPL_DBLCLK

Sent to notify **CPlApplet** that the user has chosen the icon associated with a given dialog box. **CPlApplet** should display the corresponding dialog box and carry out any user-specified tasks.

CPL_EXIT

Sent after the last **CPL_STOP** message and immediately before the controlling application uses the **FreeLibrary** function to free the DLL containing the Control Panel application. **CPlApplet** should free any remaining memory and prepare to close.

CPL_GETCOUNT

Sent after the **CPL_GETCOUNT** message to prompt **CPlApplet** to return a number indicating how many dialog boxes it supports.

CPL_INIT

Sent immediately after the DLL containing the Control Panel application is loaded, to prompt **CPlApplet** to perform initialization procedures, including memory allocation.

CPL_INQUIRE

Sent after the **CPL_GETCOUNT** message, to prompt **CPlApplet** to provide information about a specified dialog box. The lParam2 parameter of **CPlApplet** points to a TCPLInfo structure.

CPL_NEWINQUIRE

Sent after the **CPL_GETCOUNT** message, to prompt **CPlApplet** to provide information about a specified dialog box. The lParam2 parameter is a pointer to a TNewCPLInfo structure. For better performance on Windows 95 and Windows NT version 4.0, your application should process **CPL_INQUIRE** and not **CPL_NEWINQUIRE**.

CPL_SELECT

This message is obsolete. Current versions of Windows do not send this message.

CPL_STOP

Sent once for each dialog box before the controlling application closes. **CPlApplet** should free any memory associated with the given dialog box.

You will find an example of Control Panel Application in **/TMTPL/SAMPLES/WIN32/CPL** subdirectory.

Appendix A

Compiler Directives

Compiler directives, are comments started with the $ symbol. Compiler directives can be used wherever comments are allowed.

Compiler directives
- begin with **{$**, **/*$** or **(*$**
- are followed by the name of the directive
- end with **}**, ***/** or ***)**

Note that // and -- comments can not be used to specify compiler directive
Compiler directives come in three varieties:
Switch directives turn compiler features on or switches off when + or - are specified after the directive name.
Parameter directives specify parameters that affect the compilation.
Conditional directives control conditional compilation of parts of the source text.

A.1 Conditional directives

Targets: MS-DOS, OS/2, Win32

Conditional compilation is based on the evaluation of conditional symbols.

$DEFINE	Defines a conditional symbol
$ELSE	Compiles or ignores a portion of source text
$ENDIF	Ends the conditional compilation
$IFDEF	Compiles source text if Name is defined
$IFNDEF	Compiles source text if Name is NOT defined
$IFOPT	Compiles source text if a compiler switch is in a specifies state(+ or -)
$UNDEF	Undefines a previously defined conditional symbol

A.2 Switch and Parameter Directives

Targets: MS-DOS, OS/2, Win32

$A: Data Align Switch

Switches on/off word-alignment of variables and typed constants

Syntax:

```
{$A+} or {$A-}
```

Default:

```
{A+}
```

Remarks:

The data align switch has no affect on structures and objects alignment. Use $OA compiler directive to switch on/off structures and objects alignment.

$AC: Ada-Style Comments Switch

Switches on/off Ada-style comments recognition.

Syntax:

```
{$AC+} or {$AC-}
```

Default:

```
{AC-}
```

Remarks:

Keep in mind that Ada-style comments are not longer supported by default.

$AMD: AMD 3DNow! Assembler Instructions Switch

Enables/disables AMD 3DNow! instructions support in built-in assembler.

Syntax:

```
{$AMD+} or {$AMD-}
```

Default:

```
{AMD+}
```

$B: Boolean Evaluation Switch

Switches on/off the two different models of code generation for the AND and OR Boolean operators.

Syntax:

```
{$B+} or {$B-}
```

Default:

```
{$B-}
```

Remarks:

If `{$B+}` defined, the compiler generates code for complete boolean expression evaluation. I.e. every operand of a boolean expression built from the AND and OR operators is guaranteed to be evaluated, even when the result of the entire expression is already known.

If `{$B-}` defined, the compiler generates code for short-circuit boolean-expression evaluation. I.e. evaluation stops as soon as the result of the entire expression becomes evident.

$CC: C/C++ Style Comments Switch

Switches on/off C/C++ style comments recognition.

Syntax:

```
{$CC+} or {$CC-}
```

Default:

```
{CC+}
```

$D: Debug Information Switch

Switches on/off the generation of debug information.

Syntax:

```
{$D+} or {$D-}
```

Default:

```
{$D+}
```

Remarks:

Debug information consists of a line-number table for each procedure. The table maps object-code addresses into source-text line numbers.

If `{$D+}` is defined, you can use the built-in Debugger to single-step, step over and set breakpoints in a module.

Debug information increases the size of unit files and increases memory usage when you compile programs that use the unit.

The Debug Information switch is usually used with the Local Symbols switch.

See also: **$L: Local Symbol Information Switch.**

$I: I/O-Checking Switch

Enables or disables the automatic code generation that checks the result of a call to an I/O procedure.

Syntax:

```
{$I+} or {$I-}
```

Default:

```
{$I+}
```

Remarks:

If an I/O procedure returns a non-zero I/O result when the $I switch is on, the program terminates, displaying a run-time error message.

When the $I switch is off, you should use the **IOResult** function to check for I/O errors.

$Include File Directive

Instructs the compiler to include the named file in the compilation.

Syntax:

```
{$I FileName}
```

Remarks:

The included file is inserted in the compiled text right after the `{$I FileName}` directive.

$L: Link Object File Directive

Instructs the compiler to link the named file with the program or unit being compiled.

Syntax:

```
{$L FileName}
```

Remarks:

The {$L FileName} directive is used to link with code written in assembly language for sub-programs which are declared to be external.

The named file must be an Intel relocatable object file (.OBJ file).

See also: **External Declaration**

$L: Local Symbol Information Switch

Enables or disables the generation of local symbol information.

Syntax:

{$L+} or {$L-}

Default:

```
{$L+}
```

Remarks:

Local symbol information consists of the symbols in the module's implementation part (names and types of all local variables and constants in a module), and the symbols within the module's procedures and functions.

$MAP: Map File Generation Switch

Switches on/off warnings generation.

Syntax:

```
{$MAP+} or {$MAP-}
```

Default:

```
{MAP-}
```

Remarks:

If {$MAP+} defined, TMT Pascal will generate a map file.

$MMX: Intel MMX Assembler Instructions Switch

Enables/disables Intel MMX instructions support in built-in assembler.

Syntax:

```
{$MMX+} or {$MMX-}
```

Default:

```
{MMX+}
```

$OA: Objects and Structures Align Switch

Switches on/off word-alignment of objects and structures.

Syntax:

```
{$OA+} or {$OA-}
```

Default:

```
{OA-}
```

Remark:

The data align switch has no affect on variables and typed constants alignment. Use $A compiler directive to switch on/off variables and typed constants alignment.

$OPT: Full Optimization Switch

Switches on/off full optimization ({OPTREG+} & {OPTFRM+}).

Syntax:

```
{$OPT+} or {$OPT-}
```

Default:

```
{OPT+}
```

$OPTFRM: Stack Frame Optimization Switch

Switches on/off stack frame optimization.

Syntax:

```
{$OPTFRM+} or {$OPTFRM-}
```

Default:

```
{OPTFRM+}
```

$OPTREG: Register Optimization Switch

Switches on/off register optimization.

Syntax:

```
{$OPTREG+} or {$OPTREG-}
```

Default:

```
{OPTREG+}
```

$P: Open String Parameters Switch

Controls the meaning of variable parameters declared using the string keyword.

Syntax:

```
{$P+} or {$P-}
```

Default:

```
{$P+}
```

Remarks:

If `{$P-}` defined, variable parameters declared using the string keyword are normal variable parameters.

If `{$P+}` defined, variable parameters declared using the string keyword are open string parameters.

$Q: Overflow Checking Switch

Controls the generation of overflow checking code.

Syntax:

```
{$Q+} or {$Q-}
```

Default:

```
{$Q-}
```

Remarks:

The $Q switch is usually used in conjunction with the $R switch.

Enabling overflow checking slows down your program and makes it larger. We recommend to use `{$Q+}` only for debugging purposes.

$R: Range-Checking Switch

Enables and disables the generation of range-checking code.

Syntax:

```
{$R+} or {$R-}
```

Default:

```
{$R-}
```

Remarks:

The $R switch is usually used in conjunction with the $Q switch.

If `{$R+}` defined, all array and string-indexing expressions are verified as being within the defined bounds all assignments to scalar and subrange variables are checked to be within range.

If a range-check fails, the program terminates and displays a run-time error message.

Enabling range-checking slows down your program and makes it larger. We recommend to use `{$R+}` only for debugging purposes.

Keep in mind that range-checking mode affects even on «**+**», «*****» and *Shl* operators.

$R: Resource File

Targets: OS/2, Win32

Specifies the name of a resource file to be included in an application or library. The named file must be valid resource file (Windows 32-bit or OS/2 format) and the default extension for filenames is .RES.

Syntax:

```
{$R filename.RES}
```

Remarks:

When a {$R filename} directive is used in a unit, the specified file name is simply recorded in the resulting unit file. No checks are made at that point to ensure that the filename is correct and that it specifies an existing file.

Win32 target:

The old 16-bit Windows resource format is not allowed.

$S: Stack-Overflow Checking Switch

Enables and disables the generation of stack-overflow checking code.

Syntax:

```
{$S+} or {$S-}
```

Default:

```
{$S-}
```

Remarks:

If {$S+} is defined, the compiler generates code at the beginning of each procedure or function to check whether there is sufficient stack space for the local variables and other temporary storage.

Important! This option is not supported by the current version of the compiler and will be ignored.

$T: Type-Checked Pointers Switch

Controls the types of pointer values generated by the @ operator.

Syntax:

```
{$T+} or {$T-}
```

Default:

```
{$T-}
```

Remarks:

If {$T-} is defined, the resulting type of the @ operator is always an untyped pointer. Otherwise the type of the result is ^T, where T is compatible only with other pointers to the type of the variable.

$TPO: Typed Inc/Dec Operations Switch

Enables/disables typed Inc/Dec operations on pointers.

Syntax:

```
{$TPO+} or {$TPO-}
```

Default:

```
{$T+}
```

Example:

```
var
  a: ^DWORD;
begin
  a := Pointer(0);
  inc(a);
  Writeln(Longint(a));
end.
```

The sample above prints 1 if typed operations are disabled ($TPO-}. If typed operations are enabled ($TPO+}, the application prints 4.

$V: Var-String Checking Switch

Controls type-checking on strings passed as variable parameters.

Syntax:

```
{$V+} or {$V-}
```

Default:

```
{$V+}
```

Remark:

If {$V+} is defined, strict type-checking is performed, requiring the formal and actual parameters to be of identical string types. Otherwise any string-type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter.

$W: Warnings Generation Switch

Switches on/off warnings generation.

Syntax:

```
{$W+} or {$W-}
```

Default:

```
{W+}
```

Remark:

If {$W+} defined, TMT Pascal will show compilation warnings.

$X: Extended Syntax Switch

Enables or disables Turbo Pascal's extended syntax.

Syntax:

```
{$X+} or {$X-}
```

Default:

```
{$X+}
```

Remarks:

If {$X+} is defined, function calls can be used as statements. The result of a function call can be discarded.

A.3 Predefined Symbols

Targets: MS-DOS, OS/2, Win32

The following symbols are predefined:

MSDOS	- for DOS target
__TMT__	- always
__VER3__	- always for version 3.xx
__MULTITARGET__	- always for TMT Pascal multi-target
__DOS__	- for DOS target
__OS2__	- for all OS/2 targets
__DLL__	- for OS/2 and Win32 DLL targets
__PM__	- for OS/2 Presentation manager targets
__FS__	- for OS/2 Full Screen targets
__WIN32__	- for all Win32 targets
__CON__	- for OS/2, Win32 and MS-DOS console targets
__GUI__	- for Win32 GUI targets

Appendix B

Run-time Error Codes

The following error codes are predefined:

Code	Meaning
1	Invalid function number
2	File not found
3	Path not found
4	Too many open files
5	File access denied
6	Invalid file handle
12	Invalid file access code
15	Invalid drive number
16	Cannot remove current directory
17	Cannot rename across drives
18	No more files
100	Disk read error
101	Disk write error
102	File not assigned
103	File not open
104	File not open for input
105	File not open for output
106	Invalid numeric format
150	Disk is write protected
151	Bad drive request structure length
152	Drive not ready
154	CRC error in data
156	Disk seek error
157	Unknown media type
158	Sector not found
159	Printer out of paper
160	Device write fault
161	Device read fault
162	Hardware failure
200	Division by zero
201	Range check error
202	Stack overflow error
203	Heap overflow error
204	Invalid pointer operation
205	Floating point overflow
206	Floating point underflow
207	Invalid floating point operation
208	Overlay manager not installed
209	Overlay file read error
210	Object not initialized
211	Call to abstract method
212	Stream registration error
213	Collection index out of range

214	Collection overflow error
215	Arithmetic overflow error
216	General protection fault
217	Invalid operation code
300	File IO error
301	Non-matched array bounds
302	Non-local procedure pointer
303	Procedure pointer out of scope
304	Function not implemented
305	Breakpoint error
306	Break by Ctrl/C
307	Break by Ctrl/Break
308	Break by other process
309	No floating point coprocessor

Appendix C

PMODE/W DOS Extender

TMT Pascal uses the PMODEW v1.33 based extender. This chapter of the HELP file is based on the original manual

C.1 About PMODE/W

PMODE/W allows DOS programs to run in full 32 bit protected mode, with access to all memory available in the system. PMODE/W basically extends the DOS environment to protected mode and provides a simple interface to the real mode DOS system services for your code. PMODE/W takes care of all aspects of running the system in protected mode and maintaining compatibility with the underlying real mode software. PMODE/W deals with low level necessities such as descriptor tables, memory management, IRQ and interrupt redirection, real/protected mode translation functions, exception handling, and other miscellaneous aspects of running in protected mode. Your code does not need to deal with specific aspects of different systems, such as XMS/EMS/VCPI/DPMI availability. PMODE/W will run on top of almost any system and provide common protected mode services to your program through the DPMI interface specification, as well as most standard DOS functions extended for protected mode use.

PMODE/W is the stub and extender in one. The generated executable contains the PMODE/W extender within it as the stub. When run, PMODE/W will take care of setting up the system and executing the protected mode portion of the program. Several years have gone into the development of PMODE/W. It is now a fairly mature DOS extender, and has gone through its fair share of bugs and incompatibilities. It is at this point, a very stable protected mode system. Great pains have gone into the optimization and testing of PMODE/W. Our major goals have been speed, size, and stability. We now feel that we have achieved a good deal of those things. But don't take our word for it; try it yourself. Just plug PMODE/W into any popular program which uses DOS/4GW.

To sum it up, if you are looking for a good solid, stable, and fast extender, PMODE/W may be just what you need.

Here are the advantages of PMODE/W:

No external extender required (everything needed to execute is in the EXE). Small size (less than 12k for the entire extender program). Compression of protected mode executables. Low extended memory overhead. Does not require ANY extended memory to load OR execute. Fast execution.

Our major concerns in developing PMODE/W were speed, size, and stability. PMODE/W itself was written entirely in assembly. When running under PMODE/W, your code will be running at a privilege level of zero, the highest and fastest. PMODE/W does not virtualize what it does not need to, and does not invoke any protected mode mechanism that is slow. For example, if the system is running clean or under XMS, PMODE/W does not turn on paging. Under a memory manager which provides both VCPI and DPMI services, PMODE/W will

opt for VCPI protected mode which is significantly faster than DPMI. When PMODE/W makes calls to real mode, it switches the system into actual real mode rather than the slower V86 mode (when it can, under VCPI this is not possible, control must be passed back to the VCPI server). In terms of speed, when your code is running under PMODE/W, it is running as fast as the system will allow. In terms of size on disk, we need say no more than for you to look at the size of the PMODE/W executable and compare it to other extenders. In terms of memory size, you may do tests yourself to confirm that PMODE/W does indeed suck up a lot less memory at run-time than the competition. In fact, PMODE/W will run even if there is absolutely no extended memory in the system (assuming of course there is enough low memory for the program). To be fair, we must say that we squished the PMODE/W executable with our own compression program written expressly for the purpose. This demonstrates the extent we took most of our optimizations to.

When run under a clean system, XMS, or VCPI, PMODE/W has control of protected mode. In this case, it can set up the system to run as fast as possible under the various conditions. Under DPMI, the DPMI host of the system will have full protected mode control and PMODE/W will install its DOS extensions on top of that. If the system provides both VCPI and DPMI services, PMODE/W will use the VCPI services for faster execution, unless instructed not to by the setup program. When PMODE/W does have protected mode control under clean/XMS/VCPI, it runs all code at a privilege level of zero. In addition, under a clean or XMS system, paging will not be enabled. This is only a minor speed increase, but there is no real need to manage paging.

PMODE/W provides a subset of DPMI 0.9 function calls and general functionality when a DPMI host is not present. PMODE/W will pass any software interrupts from protected mode to their default real mode handlers, provided no protected mode handlers have been installed for them, just as DPMI will. The general registers will be passed on to the real mode handler, but the segment registers cannot be as they have different meanings in real mode and protected mode. The flags will be passed back from the real mode handler. This provides a simple interface to all real mode interrupt routines which do not take parameters in the segment registers, for example, INT 16h function 00h.

Any IRQs that occur in protected mode and have not been hooked by a protected mode handler will be sent on to their real mode handlers. If an IRQ occurs in real mode, and a protected mode handler has hooked that IRQ, it will be sent to the protected mode handler first. The protected mode handler may chain to the real mode handler for that IRQ by calling the previous protected mode handler for that IRQ. This behavior is in accordance with the DPMI standard. If you hook a protected mode IRQ (INT 31h function 0205h), then hook the same IRQ in real mode (INT 31h function 0201h), the protected mode handler will be called if the IRQ occurs in protected mode, and the real mode handler will handle the IRQs if they occur in real mode. Setting up two handlers like this assures minimal latency. This means a handler will get control when the IRQ occurs as soon as physically possible. PMODE/W does have to intervene in the IRQ process, however, when the low 8 IRQs are mapped to INTs 08h-15h to differentiate IRQs from CPU exceptions.

In accordance with DPMI specifications, PMODE/W will pass up software interrupts 1ch (BIOS timer tick), 23h (DOS CTRL+C), and 24h (DOS critical error) from real mode to protected mode. This means that those interrupts can be hooked directly in protected mode without having to set up a callback mechanism yourself. PMODE/W will also pass interrupt 1bh (BIOS CTRL+BREAK) from real mode up to protected mode. This is not a DPMI requirement, but it is necessary for the sake of compatibility with DOS/4GW. Another departure by PMODE/W from official DPMI specifications is in extended memory allocation. DPMI documentation states that the block of extended memory allocated through function 0501h is guaranteed at least paragraph alignment. The PMODE/W DPMI implementation will enforce only DWORD alignment.

When a PMODE/W executable is run, PMODE/W will attempt to switch the system into protected mode and load the protected mode portion of the same executable. If there is some error, not enough memory, or a system incompatibility, PMODE/W will exit with an error message. If loading was successful, PMODE/W will pass execution control on to the program.

PMODE/W will load any 16 bit code and data into low memory, but 32 bit code and data may be loaded into low or extended memory depending on availability.

There are a number of modifiable parameters in the PMODE/W extender executable that affect protected mode execution. For the most part, these parameters deal with memory. PMODE/W allocates one large block of extended memory for its pool from which it provides memory to its client program. There is a maximum value for the extended memory to be allocated. By default, the maximum is all of the extended memory in the system. The maximum value reflects the size of the block you want PMODE/W to take from the system, not necessarily the size of the largest block available to the default *GetMem* procedures. You may set the maximum to zero to indicate you do not want PMODE/W to allocate ANY extended memory. The amount of memory that you allow PMODE/W to allocate from the system determines how much extended memory will be left to other if you shell out of your PMODE/W program.

Another variable specifies the amount of low memory for PMODE/W to TRY to keep free. If PMODE/W can, it will accommodate this value by loading 32 bit code and data into extended memory. If there is not enough extended memory available for this, 32 bit code and data will be loaded into low memory anyway. If PMODE/W can not keep this much low memory free, it will not exit with an error message. Setting this parameter to a high value will, in effect, duplicate the DOS/4GW behavior of loading all 32 bit code and data into extended memory. If you do not necessarily need any extra low memory free during the execution of your program, you may set this value to zero.

There is a group of parameters that specify the number and size of nested mode switch stacks. Whenever you make a call to real mode, or a callback or IRQ is passed from real mode to its routine in protected mode, a nested stack is used. These parameters have meaning only if the program is not run under a DPMI system. If a DPMI host is in place when the program is run, it provides its own nested stacks for mode switches. The number of nested stacks directly affects the number of nested mode switches your program can make using the various mode switch methods. The size of both the real mode and protected mode nested stacks can also be specified. By default, these values are high enough for normal operation. However, if you intend to use a lot of stack variables in a protected mode IRQ handler, or a lot of recursive calls, you may need to increase the size of the protected mode nested stacks. The more nested stacks you specify, and the larger they are, the more low memory is needed by PMODE/W during execution.

Another group of variables that has meaning only under clean/XMS/VCPI execution specify the number of selectors and DPMI callbacks you want PMODE/W to make available. The more selectors and callbacks you ask for, the more low memory is used by PMODE/W, though the amount of low memory used for each is quite low so that large numbers of each can be specified. There will usually be a little less than the number of selectors and callbacks you request available to your program due to the protected mode system and Pascal code using some of them. For this reason you should request 20h-40h more selectors and 2-4 more callbacks than you will need in your program.

There are four other miscellaneous parameters that can be set. There is a maximum number of page tables to use under a VCPI system. Each page table allocated requires 4k of low memory to be used by PMODE/W and maps 4M of memory. This directly affects the maximum amount of extended memory available under a VCPI system. This parameter is only the maximum number of page tables to allow. At run-time, only as many page tables will be allocated as are needed. Under a clean/XMS system, no page tables are required, so this parameter has no meaning. But under VCPI, you may want to restrict the number of page tables to save low memory if you do not need more than a certain amount of extended memory. This puts a maximum ceiling on extended memory under VCPI which may be lower than the maximum actually specified in the other variable. The second parameter specifies the order of DPMI and VCPI detection. By default, VCPI will be checked before DPMI, but you may set DPMI to be checked before VCPI so that under a system which supports both VCPI and DPMI, DPMI will be used. The third variable specifies how many pages to reserve for physical address mapping calls (INT 31h function 0800h) under VCPI. Under XMS or a clean system, paging is not enabled, and PMODE/W does not need pages for physical address mapping. Each page will allow you to map up to 4M of address space and takes up 4k of

extended memory. So for example, if you intend to map a 2M frame buffer of a video card, you will need only one page. You may set this parameter to zero if you do not intend to map any physical addresses. The fourth parameter specifies whether PMODE/W displays its banner at startup. This may be desirable to indicate that the program is indeed running, and has not crashed, as allocating memory from certain VCPI servers can be a slow process.

C.2 Supported DPMI INT 31h functions

PMODE/W duplicates a subset of DPMI protected mode functions. These functions are available ONLY in protected through INT 31h. They provide descriptor services, extended memory services, interrupt services, translation services, and some other miscellaneous things. A function is called by setting AX to the function code, setting any other registers for the function, and executing INT 31h. Upon return, the carry flag will be clear if the function was successful. If the carry flag is set, the function failed. All other registers are preserved unless otherwise specified. In addition to the functions listed here, functions 0600h, 0601h, 0702h, and 0703h will return with the carry flag clear to stay compatible with code that uses those particular DPMI functions.

Function 0000h - Allocate Descriptors

Allocates one or more descriptors in the client's descriptor table. The descriptor(s) allocated must be initialized by the application with other function calls.

In:

```
AX =    0000h
CX = number of descriptors to allocate
```

Out:

```
if successful:
  carry flag clear
  AX     = base selector
if failed:
  carry flag set
```

Remarks:

If more that one descriptor is requested, the function returns a base selector referencing the first of a contiguous array of descriptors. The selector values for subsequent descriptors in the array can be calculated by adding the value returned by INT 31h function 0003h.
The allocated descriptor(s) will be set to expand-up writeable data, with the present bit set and a base and limit of zero. The privilege level of the descriptor(s) will match the client's code segment privilege level.

Function 0001h - Free Descriptor

Frees a descriptor.

In:

```
  AX =    0001h
  BX =    selector for the descriptor to free
```

Out:

```
if successful:
  carry flag clear
```

```
if failed:
  carry flag set
```

Remarks:

Each descriptor allocated with INT 31h function 0000h must be freed individually with the function. Even if it was previously allocated as part of a contiguous array of descriptors. Under DPMI 1.0/VCPI/XMS/raw, any segment registers which contain the selector being freed are zeroed by this function.

Function 0002h - Segment to Descriptor

Converts a real mode segment into a protected mode descriptor.

In:

```
  AX =    0002h
  BX =    real mode segment
```

Out:

```
if successful:
  carry flag clear
  AX =  selector
if failed:
  carry flag set
```

Remarks:

Multiple calls for the same real mode segment return the same selector. The returned descriptor should never be modified or freed.

Function 0003 - Get Selector Increment Value

The Allocate Descriptors function (0000h) can allocate an array of contiguous descriptors, but only return a selector for the first descriptor. The value returned by this function can be used to calculate the selectors for subsequent descriptors in the array.

In:

AX = 0003h

Out:

```
always successful:
  carry flag clear
  AX =  selector increment value
```

Remarks:

The increment value is always a power of two.

Function 0006 - Get Segment Base Address

Returns the 32bit linear base address from the descriptor table for the specified segment.

In:

```
  AX =     0006h
  BX =     selector
```

Out:

```
if successful:
```

```
  carry flag clear
  CX:DX  = 32bit linear base address of segment
if failed:
  carry flag set
```

Remarks:

Client programs must use the LSL instruction to query the limit for a descriptor.

Function 0007 - Set Segment Base Address

Sets the 32bit linear base address field in the descriptor for the specified segment.

In:

```
  AX    = 0007h
  BX    = selector
```

CX:DX = 32bit linear base address of segment

Out:

```
if successful:
  carry flag clear
if failed:
  carry flag set
```

Remarks:

Under DPMI 1.0/VCPI/XMS/raw, any segment register which contains the selector specified in register BX will be reloaded. DPMI 0.9 may do this, but it is not guaranteed. We hope you have enough sense not to try to modify your current CS or SS descriptor.

Function 0008 - Set Segment Limit

Sets the limit field in the descriptor for the specified segment.

In:

```
  AX =    0008h
  BX =    selector
    CX:DX = 32bit segment limit
```

Out:

```
if successful:
  carry flag clear
if failed:
  carry flag set
```

Remarks:

The value supplied to the function in CX:DX is the byte length of the segment-1. Segment limits greater than or equal to 1M must be page aligned. That is, they must have the low 12 bits set.

This function has an implicit effect on the “G” bit in the segment’s descriptor. Client programs must use the LSL instruction to query the limit for a descriptor.

Under DPMI 1.0/VCPI/XMS/raw, any segment register which contains the selector specified in register BX will be reloaded. DPMI 0.9 may do this, but it is not guaranteed. We hope you have enough sense not to try to modify your current CS or SS descriptor.

Function 0009 - Set Descriptor Access Rights

Modifies the access rights field in the descriptor for the specified segment.

In:

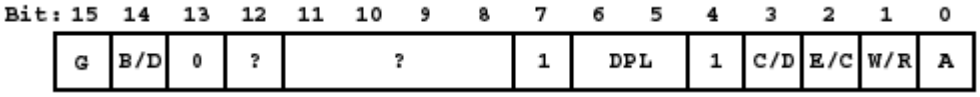
```
  AX     = 0009h
  BX     = selector
  CX     = access rights/type word
```

Out:

```
if successful:
    carry flag clear
  if failed:
    carry flag set
```

Remarks:

The access rights/type word passed to the function in CX has the following format:




```
  G   - 0=byte granular, 1=page granular
  B/D - 0=default 16bit, 1=default 32bit
  DPL - must be equal to caller's CPL
  C/D - 0=data, 1=code
  E/C - data: 0=expand-up, 1=expand-down
        code: must be 0 (non-conforming)
  W/R - data: 0=read, 1=read/write
        code: must be 1 (readable)
  A   - 0=not accessed, 1=accessed
  0   - must be 0
  1   - must be 1
  ?   - ignored
```

Client programs should use the LAR instruction to examine the access rights of a descriptor.

Under DPMI 1.0/VCPI/XMS/raw, any segment register which contains the selector specified in register BX will be reloaded. DPMI 0.9 may do this, but it is not guaranteed.

We hope you have enough sense not to try to modify your current CS or SS descriptor.

Function 000A - Create Alias Descriptor

Creates a new data descriptor that has the same base and limit as the specified descriptor.

In:

```
  AX     = 000ah
  BX     = selector
```

Out:

```
if successful:
    carry flag clear
    AX     = data selector (alias)
  if failed:
    carry flag set
```

Remarks:

The selector supplied to the function may be either a data descriptor or a code descriptor. The alias descriptor created is always an expand-up writeable data segment.

The descriptor alias returned by this function will not track changes to the original descriptor.

Function 000B - Get Descriptor

Copies the descriptor table entry for the specified selector into an 8 byte buffer.

In:

```
  AX     = 000bh
  BX     = selector
  ES:EDI = selector:offset of 8 byte buffer
```

Out:

```
  if successful:
    carry flag clear
    buffer pointed to by ES:EDI contains descriptor
  if failed:
    carry flag set
```

Function 000C - Set Descriptor

Copies the contents of an 8 byte buffer into the descriptor for the specified selector.

In:

```
  AX     = 000ch
  BX     = selector
  ES:EDI = selector:offset of 8 byte buffer containing
descriptor
```

Out:

```
  if successful:
    carry flag clear
  if failed:
    carry flag set
```

Remarks:

The descriptors access rights/type word at offset 5 within the descriptor follows the same format and restrictions as the access rights/type parameter CX to the Set Descriptor Access Rights function (0009h).

Under DPMI 1.0/VCPI/XMS/raw, any segment register which contains the selector specified in register BX will be reloaded. DPMI 0.9 may do this, but it is not guaranteed.

We hope you have enough sense not to try to modify your current CS or SS descriptor or the descriptor of the buffer.

Function 0100 - Allocate DOS Memory Block

Allocates low memory through DOS function 48h and allocates it a descriptor.

In:

```
  AX     = 0100h
```

```
  BX     = paragraphs to allocate
```

Out:

```
if successful:
    carry flag clear
    AX     = real mode segment address
    DX     = protected mode selector for memory block
  if failed:
    carry flag set
    AX     = DOS error code
    BX     = size of largest available block
```

Function 0101 - Free DOS Memory Block

Frees a low memory block previously allocated by function 0100h.

In:

```
  AX     = 0101h
  DX     = protected mode selector for memory block
```

Out:

```
  if successful:
    carry flag clear
  if failed:
    carry flag set
    AX     = DOS error code
```

Function 0102 - Resize DOS Memory Block

Resizes a low memory block previously allocated by function 0100h.

In:

```
  AX     = 0102h
  BX     = new block size in paragraphs
  DX     = protected mode selector for memory block
```

Out:

```
  if successful:
    carry flag clear
  if failed:
    carry flag set
    AX     = DOS error code
    BX     = size of largest available block
```

Function 0200 - Get Real Mode Interrupt Vector

Returns the real mode segment:offset for the specified interrupt vector.

In:

```
  AX    = 0200h
  BL    = interrupt number
```

Out:

```
always successful:
    carry flag clear
    CX:DX = segment:offset of real mode interrupt handler
```

Remarks:

The value returned in CX is a real mode segment address, not a protected mode selector.

Function 0201 - Set Real Mode Interrupt Vector

Sets the real mode segment:offset for the specified interrupt vector.

In:

```
AX     = 0201h
BL     = interrupt number
CX:DX  = segment:offset of real mode interrupt handler
```

Out:

```
always successful:
  carry flag clear
```

Remark:

The value passed in CX must be a real mode segment address, not a protected mode selector. Consequently, the interrupt handler must either reside in DOS memory (below the 1M boundary) or the client must allocate a real mode callback address.

Function 0202 - Get Processor Exception Handler Vector

Returns the address of the current protected mode exception handler for the specified exception number.

In:

```
AX     = 0202h
BL     = exception number (00h-1fh)
```

Out:

```
if successful:
  carry flag clear
  CX:EDX = selector:offset of exception handler
if failed:
  carry flag set
```

Remarks:

PMODE/W handles exceptions under clean/XMS/VCPI environments. Under a DPMI environment, exception handling is provided by the DPMI host.

PMODE/W only traps exceptions 0 through 14. The default behavior is to terminate execution and do a debug dump. PMODE/W will terminate on exceptions 0, 1, 2, 3, 4, 5, and 7, instead of passing them down to the real mode handlers as DPMI specifications state.

Function 0203 - Set Processor Exception Handler Vector

Sets the address of a handler for a CPU exception or fault, allowing a protected mode application to intercept processor exceptions.

In:

```
AX     = 0203h
BL     = exception number (00h-1fh)
```

```
  CX:EDX = selector:offset of exception handler
```

Out:

```
  if successful:
    carry flag clear
  if failed:
    carry flag set
```

Remarks:

PMODE/W handles exceptions under clean/XMS/VCPI environments. Under a DPMI environment, exception handling is provided by the DPMI host.

PMODE/W only traps exceptions 0 through 14. The default behavior is to terminate execution and do a debug dump. PMODE/W will terminate on exceptions 0, 1, 2, 3, 4, 5, and 7, instead of passing them down to the real mode handlers as DPMI specifications state.

If you wish to hook one of the low 8 interrupts, you must hook it as an exception. It will not be called if you hook it with function 0205h.

Function 0204 - Get Protected Mode Interrupt Vector

Returns the address of the current protected mode interrupt handler for the specified interrupt.

In:

```
  AX     = 0204h
  BL     = interrupt number
```

Out:

```
always successful:
    carry flag clear
    CX:EDX = selector:offset of protected mode interrupt
handler
```

Remarks:

The value returned in CX is a valid protected mode selector, not a real mode segment address.

Function 0205 - Set Protected Mode Interrupt Vector

Sets the address of the protected mode interrupt handler for the specified interrupt.

In:

```
  AX     = 0205h
  BL     = interrupt number
  CX:EDX = selector offset of protected mode interrupt handler
```

Out:

```
  if successful:
    carry flag clear
  if failed:
    carry flag set
```

Remarks:

The value passed in CX must be a valid protected mode selector, not a real mode segment address.

If you wish to hook one of the low 8 interrupts, you must hook it as an exception. It will not be called if you hook it with function 0205h.

Function 0300 - Simulate Real Mode Interrupt

Simulates an interrupt in real mode. The function transfers control to the address specified by the real mode interrupt vector. The real mode handler must return by executing an IRET.

In:

```
  AX     = 0300h
  BL     = interrupt number
  BH     = must be 0
  CX     = number of words to copy from the protected mode
stack to the real mode stack
ES:EDI   = selector:offset of real mode register data structure
in the following format:
```

Offset	Length	Contents
00h	4	EDI
04h	4	ESI
08h	4	EBP
0ch	4	reserved, ignored
10h	4	EBX
14h	4	EDX
18h	4	ECX
1ch	4	EAX
20h	2	CPU status flags
22h	2	ES
24h	2	DS
26h	2	FS
28h	2	GS
2ah	2	IP (reserved, ignored)
2ch	2	CS (reserved, ignored)
2eh	2	SP
30h	2	SS

Out:

```
  if successful:
    carry flag clear
    ES:EDI = selector offset of modified real mode register
data structure
  if failed:
    carry flag set
```

Remarks:

The CS:IP in the real mode register data structure is ignored by this function. The appropriate interrupt handler will be called based on the value passed in BL.

If the SS:SP fields in the real mode register data structure are zero, a real mode stack will be provided by the host. Otherwise the real mode SS:SP will be set to the specified values before the interrupt handler is called.

The flags specified in the real mode register data structure will be put on the real mode interrupt handler's IRET frame. The interrupt handler will be called with the interrupt and trace flags clear.

Values placed in the segment register positions of the data structure must be valid for real mode. That is, the values must be paragraph addresses, not protected mode selectors.

The target real mode handler must return with the stack in the same state as when it was called. This means that the real mode code may switch stacks while it is running, but must return on the same stack that it was called on and must return with an IRET.

When this function returns, the real mode register data structure will contain the values that were returned by the real mode interrupt handler. The CS:IP and SS:SP values will be unmodified in the data structure.

It is the caller's responsibility to remove any parameters that were pushed on the protected mode stack.

Function 0301 - Call Real Mode Procedure With Far Return Frame

Simulates a FAR CALL to a real mode procedure. The called procedure must return by executing a RETF instruction.

In:

```
  AX     = 0301h
  BH     = must be 0
  CX     = number of words to copy from the protected mode
stack to the real mode stack
ES:EDI   = selector:offset of real mode register data structure
in the following format:
```

Offset	Length	Contents
00h	4	EDI
04h	4	ESI
08h	4	EBP
0ch	4	reserved, ignored
10h	4	EBX
14h	4	EDX
18h	4	ECX
1ch	4	EAX
20h	2	CPU status flags
22h	2	ES
24h	2	DS
26h	2	FS
28h	2	GS
2ah	2	IP
2ch	2	CS
2eh	2	SP
30h	2	SS

Out:

```
  if successful:
    carry flag clear
    ES:EDI = selector offset of modified real mode register
data structure
  if failed:
    carry flag set
```

Remarks:

The CS:IP in the real mode register data structure specifies the address of the real mode procedure to call.

If the SS:SP fields in the real mode register data structure are zero, a real mode stack will be provided by the host. Otherwise the real mode SS:SP will be set to the specified values before the procedure is called.

Values placed in the segment register positions of the data structure must be valid for real mode. That is, the values must be paragraph addresses, not protected mode selectors.

The target real mode procedure must return with the stack in the same state as when it was called. This means that the real mode code may switch stacks while it is running, but must

return on the same stack that it was called on and must return with a RETF and should not clear the stack of any parameters that were passed to it on the stack.

When this function returns, the real mode register data structure will contain the values that were returned by the real mode procedure. The CS:IP and SS:SP values will be unmodified in the data structure.

It is the caller's responsibility to remove any parameters that were pushed on the protected mode stack.

Function 0302 - Call Real Mode Procedure With IRET Frame

Simulates a FAR CALL with flags pushed on the stack to a real mode routine. The real mode procedure must return by executing an IRET instruction or a RETF 2.

In:

```
  AX     = 0302h
  BH     = must be 0
  CX     = number of words to copy from the protected mode
stack to the real mode stack
  ES:EDI = selector:offset of real mode register data structure
in the following format:
```

Offset	Length	Contents
00h	4	EDI
04h	4	ESI
08h	4	EBP
0ch	4	reserved, ignored
10h	4	EBX
14h	4	EDX
18h	4	ECX
1ch	4	EAX
20h	2	CPU status flags
22h	2	ES
24h	2	DS
26h	2	FS
28h	2	GS
2ah	2	IP
2ch	2	CS
2eh	2	SP
30h	2	SS

Out:

```
if successful:
    carry flag clear
    ES:EDI = selector offset of modified real mode register
data structure
  if failed:
    carry flag set
```

Remarks:

The CS:IP in the real mode register data structure specifies the address of the real mode procedure to call.

If the SS:SP fields in the real mode register data structure are zero, a real mode stack will be provided by the host. Otherwise the real mode SS:SP will be set to the specified values before the procedure is called.

The flags specified in the real mode register data structure will be put on the real mode procedure's IRET frame. The procedure will be called with the interrupt and trace flags clear.

Values placed in the segment register positions of the data structure must be valid for real mode. That is, the values must be paragraph addresses, not protected mode selectors.

The target real mode procedure must return with the stack in the same state as when it was called. This means that the real mode code may switch stacks while it is running, but must return on the same stack that it was called on and must return with an IRET or discard the flags from the stack with a RETF 2 and should not clear the stack of any parameters that were passed to it on the stack.

When this function returns, the real mode register data structure will contain the values that were returned by the real mode procedure. The CS:IP and SS:SP values will be unmodified in the data structure.

It is the caller's responsibility to remove any parameters that were pushed on the protected mode stack.

Function 0303 - Allocate Real Mode Callback Address

Returns a unique real mode segment:offset, known as a "real mode callback", that will transfer control from real mode to a protected mode procedure. Callback addresses obtained with this function can be passed by a protected mode program to a real mode application, interrupt handler, device driver, TSR, etc... so that the real mode program can call procedures within the protected mode program.

In:

```
AX     = 0303h
  DS:ESI = selector:offset of protected mode procedure to call
  ES:EDI = selector:offset of 32h byte buffer for real mode
register data structure to be used when calling the callback
routine.
```

Out:

```
if successful:
    carry flag clear
    CX:DX  = segment:offset of real mode callback
  if failed:
    carry flag set
```

Remarks:

A descriptor may be allocated for each callback to hold the real mode SS descriptor. Real mode callbacks are a limited system resource. A client should release a callback that it is no longer using.

Function 0304 - Free Real Mode Callback Address

Releases a real mode callback address that was previously allocated with the Allocate Real Mode Callback Address function (0303h).

In:

```
AX     = 0304h
  CX:DX  = segment:offset of real mode callback to be freed
```

Out:

```
if successful:
    carry flag clear
  if failed:
    carry flag set
```

Remark:

Real mode callbacks are a limited system resource. A client should release any callback that it is no longer using.

Function 0305 - Get State Save/Restore Addresses

Returns the address of two procedures used to save and restore the state of the current task's registers in the mode (protected or real) which is not currently executing.

In:

```
    AX     = 0305h
```

Out:

```
  always successful:
    carry flag clear
    AX     = size of buffer in bytes required to save state
    BX:CX  = segment:offset of real mode routine used to
save/restore state
    SI:EDI = selector:offset of protected mode routine used to
save/restore state
```

Remarks:

_ The real mode segment:offset returned by this function should be called only in real mode to save/restore the state of the protected mode register The protected mode selector:offset returned by this function should be called only in protected mode to save/restore the state of the real mode registers.

Both of the state save/restore procedures are entered by a FAR CALL with the following parameters:

```
  AL       = 0 to save state
           = 1 to restore state
  ES:(E)DI = (selector or segment):offset of state buffer
```

The state buffer must be at least as large as the value returned in AX by INT 31h function 0305h. The state save/restore procedures do not modify an registers. DI must be used for the buffer offset in real mode, EDI must be used in protected mode.

Some DPMI hosts and VCPI/XMS/raw will not require the state to be saved, indicating this by returning a buffer size of zero in AX. In such cases, that addresses returned by this function can still be called, although the will simply return without performing any useful function.

Clients do not need to call the state save/restore procedures before using INT 31h function 0300h, 0301h, or 0302h. The state save/restore procedures are provided for clients that use the raw mode switch services only.

Function 0306 - Get Raw Mode Switch Addresses

Returns addresses that can be called for low level mode switching.

In:

```
    AX     = 0306h
```

Out:

```
  always successful:
    carry flag clear
    BX:CX  = segment:offset of real to protected mode switch
procedure
```

```
    SI:EDI = selector:offset of protected to real mode switch
procedure
```

Remarks:

The real mode segment:offset returned by this function should be called only in real mode to switch to protected mode. The protected mode selector:offset returned by this function should be called only in protected mode to switch to real mode.

The mode switch procedures are entered by a FAR JMP to the appropriate address with the following parameters:

```
  AX    = new DS
  CX    = new ES
  DX    = new SS
  (E)BX = new (E)SP
  SI    = new CS
  (E)DI = new (E)IP
```

The processor is placed into the desired mode, and the DS, ES, SS, (E)SP, CS, and (E)IP registers are updated with the specific values. In other words, execution of the client continues in the requested mode at the address provided in registers SI:(E)DI. The values specified to be placed into the segment registers must be appropriate for the destination mode. That is, segment addresses for real mode, and selectors for protected mode.

The values in EAX, EBX, ECX, EDX, ESI, and EDI after the mode switch are undefined. EBP will be preserved across the mode switch call so it can be used as a pointer. FS and GS will contain zero after the mode switch.

If interrupts are disabled when the mode switch procedure is invoked, they will not be re-enabled by the host (even temporarily). It is up to the client to save and restore the state of the task when using this function to switch modes. This requires the state save/restore procedures whose addresses can be obtained with INT 31h function 0305h.

Function 0400 - Get Version

Returns the version of the DPMI Specification implemented by the DPMI host. The client can use this information to determine what functions are available.

In:

```
    AX     = 0400h
```

Out:

```
  always successful:
    carry flag clear
    AH   = DPMI major version as a binary number (VCPI/XMS/raw
returns 00h)
    AL   = DPMI minor version as a binary number (VCPI/XMS/raw
returns 5ah)
    BX   = flags:
           Bits Significance
           0    0 = host is 16bit (PMODE/W never runs under one
of these)
                1 = host is 32bit
           1    0 = CPU returned to V86 mode for reflected
interrupts
                1 = CPU returned to real mode for reflected
interrupts
           2    0 = virtual memory not supported
                1 = virtual memory supported
           3-15 reserved
    CL   = processor type:
```

```
           03h = 80386
           04h = 80486
           05h = 80586
           06h-ffh = reserved
     DH   = current value of master PIC base interrupt (low 8
IRQs)
     DL   = current value of slave PIC base interrupt (high 8
IRQs)
```

Remark:

The major and minor version numbers are binary, not BCD. So a DPMI 0.9 implementation would return AH as 0 and AL as 5ah (90).

Function 0500 - Get Free Memory Information

Returns information about the amount of available memory. Since DPMI clients could be running in a multitasking environment, the information returned by this function should be considered advisory.

In:

```
  AX     = 0500h
  ES:EDI = selector:offset of 48 byte buffer
```

Out:

```
if successful:
    carry flag clear
    buffer is filled with the following information:
      Offset  Length  Contents
      00h     4       Largest available free block in bytes
      04h     2ch     Other fields only supplied by DPMI
  if failed:
    carry flag set
```

Remark:

Only the first field of the structure is guaranteed to contain a valid value. Any fields that are not supported by the host will be set to -1 (0ffffffffh) to indicate that the information is not available.

Function 0501 - Allocate Memory Block

Allocates a block of extended memory.

In:

```
  AX     = 0501h
  BX:CX  = size of block in bytes (must be non-zero)
```

Out:

```
  if successful:
    carry flag clear
    BX:CX  = linear address of allocated memory block
    SI:DI  = memory block handle (used to resize and free
block)
  if failed:
    carry flag set
```

Remarks:

The allocated block is guaranteed to have at least DWORD alignment.

This function does not allocate any descriptors for the memory block. It is the responsibility of the client to allocate and initialize any descriptors needed to access the memory with additional function calls.

Function 0502 - Free Memory Block

Frees a memory block previously allocated with the Allocate Memory Block function (0501h).

In:

```
AX     = 0502h
SI:DI  = memory block handle
```

Out:

```
if successful:
  carry flag clear
if failed:
  carry flag set
```

Remark:

No descriptors are freed by this call. It is the client's responsibility to free any descriptors that it previously allocated to map the memory block. Descriptors should be freed before memory blocks.

Function 0503 - Resize Memory Block

Changes the size of a memory block previously allocated with the Allocate Memory Block function (0501h).

In:

```
AX     = 0503h
BX:CX  = new size of block in bytes (must be non-zero)
SI:DI  = memory block handle
```

Out:

```
if successful:
  carry flag clear
  BX:CX  = new linear address of memory block
  SI:DI  = new memory block handle
if failed:
  carry flag set
```

Remarks:

After this function returns successfully, the previous handle for the memo block is invalid and should not be used anymore.

It is the client's responsibility to update any descriptors that map the memory block with the new linear address after resizing the block.

Function 0800 - Physical Address Mapping

Converts a physical address into a linear address. This functions allows the client to access devices mapped at a specific physical memory address.

In:

```
  AX     = 0800h
  BX:CX  = physical address of memory
  SI:DI  = size of region to map in bytes
```

Out:

```
  if successful:
    carry flag clear
    BX:CX  = linear address that can be used to access the
physical memory
  if failed:
    carry flag set
```

Remarks:

It is the caller's responsibility to allocate and initialize a descriptor for access to the memory.

Clients should not use this function to access memory below the 1 MB boundary.

Function 0801 - Free Physical Address Mapping

Releases a mapping of physical to linear addresses that was previously obtained with function 0800h.

In:

```
  AX     = 0801h
  BX:CX  = linear address returned by physical address mapping
call
```

Out:

```
  if successful:
    carry flag clear
  if failed:
    carry flag set
```

Remark:

The client should call this function when it is finished using a device previously mapped to linear addresses with function 0801h.

Function 0900 - Get and Disable Virtual Interrupt State

Disables the virtual interrupt flag and returns the previous state of it.

In:

```
    AX     = 0900h
```

Out:

```
  always successful:
    carry flag clear
    AL     = 0 if virtual interrupts were previously disabled
    AL     = 1 if virtual interrupts were previously enabled
```

Remarks:

AH is not changed by this function. Therefore the previous state can be restored by simply executing another INT 31h.

A client that does not need to know the prior interrupt state can execute the CLI instruction rather than calling this function. The instruction may be trapped by a DPMI host and should be assumed to be very slow.

Function 0901 - Get and Enable Virtual Interrupt State

Enables the virtual interrupt flag and returns the previous state of it.

In:

```
AX     = 0901h
```

Out:

```
  always successful:
    carry flag clear
    AL     = 0 if virtual interrupts were previously disabled
    AL     = 1 if virtual interrupts were previously enabled
```

Remarks:

AH is not changed by this function. Therefore the previous state can be restored by simply executing another INT 31h.

A client that does not need to know the prior interrupt state can execute the STI instruction rather than calling this function. The instruction may be trapped by a DPMI host and should be assumed to be very slow.

Function 0902 - Get Virtual Interrupt State

Returns the current state of the virtual interrupt flag.

In:

```
    AX     = 0902h
```

Out:

```
  always successful:
    carry flag clear
    AL     = 0 if virtual interrupts are disabled
    AL     = 1 if virtual interrupts are enabled
```

Remark:

This function should be used in preference to the PUSHF instruction to examine the interrupt flag, because the PUSHF instruction returns the physical interrupt flag rather than the virtualized interrupt flag. On some DPMI hosts, the physical interrupt flag will always be enabled, even when the hardware interrupts are not being passed through to the client.

Function EEFF - Get DOS Extender Information

Returns information about the DOS extender.

In:

```
AX     = EEFFh
```

Out:

```
  if successful:
    carry flag clear
    EAX    = 'PMDW' (504D4457h)
    ES:EBX = selector:offset of ASCIIZ copyright string
    CH     = protected mode system type (0=raw, 1=XMS, 2=VCPI,
3=DPMI)
    CL     = processor type (3=386, 4=486, 5=586)
    DH     = extender MAJOR version (binary)
    DL     = extender MINOR version (binary)
  if failed:
    carry flag set
```

Remarks:

In PMODE/W's implementation of this function, the value returned in ES is equivalent to the 4G data selector returned in DS at startup.

This function is always successful under PMODE/W.

Appendix D

IDE Overview

TMT Pascal compiler comes with an IDE for Win32, which allows one to easily edit, compile and execute applications for any target.

Features

- Tunable syntax highlighting.
- Multi-level undo buffer.
- Code templates.
- Clipboard history window.
- Comfortable multi-window editor, which allows one to edit and run sources.
- Up to 10 bookmarks.
- New Windows-based context-sensitive help system.
- ANSI/OEM character insertion table.
- Powerful search/replace engine, which allow to find specified text in open windows and directories.
- Easy in use compiler options setup menu.
- Multi-target compilation support.

Restrictions

The built-in debugger is not implemented in the current version of IDEW32.

See also: **Bookmarks, Code Templates, Compiler Options, Directories, Display, Editor, Editor Shortcuts**

D.1 Bookmarks

The Code Editor supports up to 10 bookmarks. Set your own bookmarks by right-clicking in the Code editor and choosing the toggle **Bookmarks**. To jump to a bookmark, right-click and choose **Goto Bookmarks**. You can also toggle bookmark #0 by left-clicking on the left gutter.

The following bookmark operations shortcuts are available:

Shortcut	Action
Shift+Ctrl+0	Sets bookmark 0
Shift+Ctrl+1	Sets bookmark 1
Shift+Ctrl+2	Sets bookmark 2
Shift+Ctrl+3	Sets bookmark 3
Shift+Ctrl+4	Sets bookmark 4
Shift+Ctrl+5	Sets bookmark 5
Shift+Ctrl+6	Sets bookmark 6
Shift+Ctrl+7	Sets bookmark 7
Shift+Ctrl+8	Sets bookmark 8
Shift+Ctrl+9	Sets bookmark 9

Shortcut	Action
Ctrl+0	Goes to bookmark 0
Ctrl+1	Goes to bookmark 1
Ctrl+2	Goes to bookmark 2
Ctrl+3	Goes to bookmark 3
Ctrl+4	Goes to bookmark 4
Ctrl+5	Goes to bookmark 5
Ctrl+6	Goes to bookmark 6
Ctrl+7	Goes to bookmark 7
Ctrl+8	Goes to bookmark 8
Ctrl+9	Goes to bookmark 9

D.2 Code Templates (Options | Environment | Code Templates)

A set of templates is available to insert common programming statements into your code. Templates can be edited and added. While working in the Code Editor, press *Ctrl+J* to display the code templates defined.

To edit the name and description:

Select the name you want to edit. Click the **Edit** button. Edit the name and description fields as needed and click OK.

To edit a template:

When a name is selected, the code to be inserted in the file when the template is selected is displayed in the code window. Move the cursor to the code window and edit the text as you desire.

To define the insertion point:

Place a vertical bar «|» in the code statement to define the point to begin insertion when the template is inserted in a code file. The cursor will be placed in the point defined by the vertical bar.

To add a template:

Click **Add** button. After entering a **Name** and **Description** in the dialog box displayed, click *OK*.

To delete a template:

Select the name of the template you want to delete. Click **Delete** or press *Del*.

D.3 Compiler Options (Options | Compiler)

Primary file
Specify the primary file to be used for the target executable file.

Active Window
Check this box to set a currently active window as primary file.

Word alignment data
Switches on/off word-alignment of variables and typed constants. Corresponds to (**$A**).

Strict var-strings
Controls type-checking on strings passed as variable parameters. Corresponds to (**$V**).

Range checking
Enables and disables the generation of range-checking code. Corresponds to (**$R**).

Objects and structures align
Switches on/off word-alignment of objects and structures. Corresponds to (**$OA**).

Debug information (MS-DOS target only)
Switches on/off the generation of debug information. Corresponds to (**$D**).

I/O checking
Enables or disables the automatic code generation that checks the result of a call to an I/O procedure. Corresponds to (**$I**).

Local debug symbols (MS-DOS target only)
Enables or disables the generation of local symbol information. Corresponds to (**$L**).

Open string params
Controls the meaning of variable parameters declared using string keyword. Corresponds to (**$P**).

Overflow checking (Q)
Controls the generation of overflow checking code. Corresponds to (**$Q**).

Typed pointers
Controls the types of pointer values generated by the @ operator. Corresponds to (**$T**).

Show warnings
Switches on/off warnings generation. Corresponds to (**$W**).

Extended syntax
Enables or disables Turbo Pascal's extended syntax. Corresponds to (**$X**).

Extender logo (MS-DOS target only)

Switches on/off extender logo.

Complete Boolean eval

Switches on/off the two different models of code generation for the AND and OR Boolean operators. Corresponds to (**$B**).

Frame optimization

Switches on/off stack frame optimization. Corresponds to (**$OPTFRM**).

Registers optimization

Switches on/off register optimization. Corresponds to (**$OPTREG**).

Full optimization

Switches on/off full optimization (Frame optimization + Registers optimization). Corresponds to (**$OPT**).

C/C++ style comments

Switches on/off C/C++ style comments recognition. Corresponds to (**$CC**).

Ada-style comments

Aligns elements in structures to 32-bit boundaries. Corresponds to (**$AC**).

Intel MMX Assembler instructions

Enables/disables Intel MMX instructions support in built-in assembler. Corresponds to (**$MMX**).

AMD 3DNow! Assembler instructions

Enables/disables AMD 3DNow! Instructions support in built-in assembler. Corresponds to (**$AMD**).

Typed Inc/Dec operations

Enables/disables typed Inc/Dec operations on pointers. Corresponds to (**$TPO**).

Max EXE size

Specifies the maximum size of the executable module.

Stack size

Specifies the size of the application stack.

Max OBJ size

Specifies the size of the buffer for object modules (FPD/FPO/FPW-files). This parameter must be about one and a half times the size of the largest object module from the project.

Max resource size

Specifies the size of the linking buffer for Windows resources (RES-files).

Target (Multi-target Edition Only)
A drop-down box which allows one to select a target Operating System.

D.4 Directories (Options | Directories)

Use this page to specify directories, compiler and stub names. Click the **«...»** button to run directory browse window.

Root Path
Specifies directories where the TMT Pascal compiler has been installed. Default is C:\TMTPL.

Stub name
Name of the stub file to be linked with a generated application.

Resource Compiler
Specifies a name of the command-line resource compiler.

Compiler
Specifies a name of the command-line TMT Pascal compiler.

Source Path
Specifies directories where the compiler looks for source files when it cannot find them. Default is SRC:;SYS:..\UNITS.

Unit Path
Specifies directories where the compiler looks for RTL unit files when it cannot find them. Default is SRC:;SYS:..\UNITS.

OBJ Path
Specifies directories where the compiler looks for OBJ files when it cannot find them. Default is SRC:;SYS:..\UNITS.

D.5 Display (Options | Environment | Display)

Use the Color page of the Environment Options dialog box to specify how the different elements of your code appear in the Code Editor.

You can specify foreground and background colors for anything listed in the Element list box. The sample Code Editor shows how your settings will appear in the Code Editor.

Display | General
Use the General page of the Environment Options dialog box to select display and font options for the Code Editor

Visible right margin

Check to display a line at the right margin of the Code Editor.

Right margin

Set the right margin of the Code Editor. The default is 80 characters.

Margin color

Select a color for the right margin.

Visible gutter

Check to display the gutter on the left edge of the Code Editor.

Gutter width

Set the width of the gutter in pixels.

Gutter color

Select a color for the gutter.

Left indent

Specify the left indent for the Editor.

Font name

Select a font type from the available screen fonts installed on your system (shown in the drop-down box). The Code Editor displays and uses only fixed size screen fonts. A sample text is displayed in the text box.

Font size

Select a **Font name** from the predefined font sizes associated with the font you selected in the Font list box. A sample text is displayed in the text box.

Display | Syntax Highlighting

Use the Syntax Highlighting page of the Environment Options/Display dialog box to specify the syntax highlighting scheme to be used in the Code Editor.

Element

Specifies syntax highlighting for a particular code element. You can choose from the Element list or click the element in the sample Code Editor.

Color scheme

Enables you to quickly configure the color scheme using predefined color combinations. The sample text box below shows how your settings will appear in the Code Editor. Predefined color schemes are the following: Defaults, Classic, Borland Delphi, Microsoft Visual Studio, Twilight, Ocean.

Foreground

Sets the foreground color for the selected code element.

Background
Sets the background color for the selected code element.

D.6 Editor (Options | Environment | Editor)

Use the Editor page of the Environment Options dialog box to customize the behavior of the Code Editor.

Keystroke mapping
Use the Keystroke mapping to configure the editor.

Overwrite blocks
Replaces a marked block of text with whatever is typed next. If Persistent Blocks is also selected, the text you enter is appended following the currently selected block.

Limit EOL
Limits the position of the cursor beyond the end-of-line character.

Force Cut and Copy enabled
Enables Edit|Cut and Edit|Copy, even when there is no text selected.

Undo after save
Allows you to retrieve changes after a save.

Find text at cursor
Places the text at the cursor into the Text To Find list box in the Find Text dialog box when you choose Search|Find. When this option is disabled you must type in the search text, unless the Text To Find list box is blank, in which case the editor still inserts the text at the cursor.

Highlight URLs
If URLs highlighting is enabled and if the text that is displayed in the Code Editor contains URLs then they will automatically be highlighted. Please notice that if the URL is highlighted then users can click on it and browse the corresponding Web location with the installed browser.

Double click line
Highlights the line when you double-click any character in the line. If disabled, only the selected word is highlighted.

Auto Indent mode
Positions the cursor under the first nonblank character of the preceding nonblank line when you press *Enter*.

Backspace unindents
Aligns the insertion point to the previous indentation level (outdents it) when you press *Backspace*, if the cursor is on the first nonblank character of a line.

Smart tab

Tabs to the first non-whitespace character in the preceding line. If Use Tab Character is enabled, this option is off.

Disable dragging

Disables text-dragging feature.

Persistent blocks

Keeps marked blocks selected even when the cursor is moved, until a new block is selected.

Use syntax highlighting

Enables syntax highlighting. To set highlighting options, use the Colors page

Allow overwrite caret shape

Allows overwrite caret shape, when insert mode is disabled.

Enable group undo

Undoes your last editing command as well as any subsequent editing commands of the same type, if you press *Alt+Backspace* or choose Edit|Undo.

Select text only

Limits a selection by the cursor to the end-of-line character.

Cursor beyond EOF

Positions the cursor beyond the end-of-file character.

Use system clipboard history

Allows the Code Editor to use the system clipboard.

Max horizontal pos

Specifies the maximum horizontal position in the Code Editor.

Spaces in tabs

Specifies the spaces number in the tabs.

Block indent

Specifies the number of spaces to indent a marked block. The default is 2.

Tab Stops

Sets the character columns that the cursor will move to each time you press *Tab*.

D.7 Editor Shortcuts

Shortcut	Action or command
F1, Ctrl + F1	Topic Search
F3	Search|Search Again
F6	Displays the next window
Shift+F6	Displays the previous window
Ctrl+I	Inserts a tab
Ctrl+L	Search|Search Again
Ctrl+J	Displays the code templates box
Ctrl+N	Inserts a new line
Ctrl+R	Search|Replace
Ctrl+S	File|Save
Ctrl+T	Deletes a word right
Ctrl+V	Edit|Insert
Ctrl+W	Deletes a word left
Ctrl+Y	Deletes a line
Ctrl+Z	Edit|Undo
Shift+Ctrl+Z	Edit|Redo
End	Moves to the end of a line
Home	Moves to the start of a line
Enter	Inserts a carriage return
Ins	Turns insert mode on/off
Del	Deletes the character to the right of the cursor
Backspace	Deletes the character to the left of the cursor
Tab	Inserts a tab
Space	Inserts a blank space
Left Arrow	Moves the cursor left one column, accounting for the autoindent setting
Right Arrow	Moves the cursor right one column, accounting for the autoindent setting
Up Arrow	Moves up one line
Down Arrow	Moves down one line
Page Up	Moves up one page
Page Down	Moves down one page
Ctrl+Alt+F	Searches word at cursor
Ctrl+Left Arrow	Moves one word left
Ctrl+Right Arrow	Moves one word right
Ctrl+Home	Moves to the top of a screen
Ctrl+End	Moves to the end of a screen
Ctrl+PgDn	Moves to the bottom of a file
Ctrl+PgUp	Moves to the top of a file
Ctrl+Backspace	Move one word to the right
Ctrl+Space	Inserts a blank space
Ctrl+Enter	Opens file at cursor
Ctrl+Tab	Moves to the next page
Ctrl+Shift+I	Indents Block
Ctrl+Shift+U	Outdents Block
Ctrl+Shift+Y	Deletes to the end of line
Shift+Tab	Inserts a tab

Shift+Backspace	Deletes the character to the left of the cursor
Shift+Left Arrow	Selects the character to the left of the cursor
Shift+Right Arrow	Selects the character to the right of the cursor
Shift+Up Arrow	Moves the cursor up one line and selects from the left of the starting cursor position
Shift+Down Arrow	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+PgUp	Moves the cursor up one screen and selects from the left of the starting cursor position
Shift+PgDn	Moves the cursor down one line and selects from the right of the starting cursor position
Shift+End	Selects from the cursor position to the end of the current line
Shift+Home	Selects from the cursor position to the start of the current line
Shift+Space	Inserts a blank space
Shift+Enter	Inserts a new line with a carriage return
Shift+Ctrl+Tab	Moves to the previous page
Ctrl+Shift+Left Arrow	Selects the word to the left of the cursor
Ctrl+Shift+Right Arrow	Selects the word to the right of the cursor
Ctrl+Shift+Home	Selects from the cursor position to the start of the current file
Ctrl+Shift+End	Selects from the cursor position to the end of the current file
Ctrl+Shift+PgDn	Selects from the cursor position to the bottom of the screen
Ctrl+Shift+PgUp	Selects from the cursor position to the top of the screen
Ctrl+Shift+Tab	Moves to the previous page
Alt+Backspace	Edit|Undo

Hint:

When selecting, hold *Alt* to select a vertical block.

See also: **Bookmarks**