## Memory Management

When a TURBO program is executed, three segments are allocated for the program: A code segment, a data segment, and a stack segment.

Code segment (CS is the code segment register):

```
CS:0000 - CS:00FF    MS-DOS base page.
CS:0100 - CS:EOFR    Run-time library code.
CS:EOFR - CS:EOFP    Program code.
CS:EOFP - CS:EOFC    Unused.
```

Data segment (DS is the data segment register):

```
DS:0000 - DS:EOFW    Run-time library workspace.
DS:EOFW - DS:EOFM    Main program block variables.
DS:EOFM - DS:EOFD    Unused.
```

The unused areas between (CS:EOFP-CS:EOFC and DS:EOFM-DS:EOFD) are allocated only if a minimum c**O**de segment size larger than the required size is specified at compilation. The sizes of the code and data segments never exceed 64K bytes each.

The stack segment is slightly more complicated, as it may be larger than 64K bytes. On entry to the program the stack segment register (SS) and the stack pointer (SP) is loaded so that SS:SP points at the very last byte available in the entire segment. During execution of the program SS is never changed but SP may move downwards until it reaches the bottom of the segment, or 0 (corresponding to 64K bytes of stack) if the stack segment is larger than 64K bytes.

The heap grows from low memory in the stack segment towards the actual stack residing in high memory. Each time a variable is allocated on the heap, the heap pointer (which is a double word variable maintained by the TURBO run-time system) is moved upwards, and then normalized, so that the offset address is always between $0000 and $000F. Therefore, the maximum size of a single variable that can be allocated on the heap is 65521 bytes (corresponding to $10000 less $000F). The total size of all variables allocated on the heap is however only limited by the amount of memory available. The heap pointer is available to the programmer through the *HeapPtr* standard identifier. *HeapPtr* is a typeless pointer which is compatible with all pointer types. Assignments to *HeapPtr* should be exercised only with extreme care.



# Chapter 21
# CP/M-86

This chapter describes features of TURBO Pascal specific to the CP/M-86 implementation. It presents two kinds of information:

1) Things you should know to make efficient use of TURBO Pascal. Pages 227 through 240.

2) The rest of the chapter describes things which are of interest only to experienced programmers, such as machine language routines, technical aspects of the compiler, etc.

## Compiler Options

The **O** command selects the following menu from which you may view and change some default values of the compiler. It also provides a helpful function to find runtime errors in programs compiled into object code files.

```
compile -> Memory
           Cmd-file
           cHn-file

command line Parameter:

Find run-time error  Quit
```

*Figure 21-1: Options Menu*

## *Memory / Cmd file / cHn-file*

The three commands **M**, **C**, and **H** select the compiler mode, i.e. where to put the code which results from the compilation. **M**emory is the default mode. When active, code is produced in memory and resides there ready to be activated by a **R**un command.



ClibPDF - www.fastio.com

**C**md-file is selected by pressing **C**. The arrow moves to point to this line. The compiler writes code to a file with the same name as the Work file (or Main file, if specified) and the file type .CMD. This file contains the program code and Pascal runtime library, and may be activated by typing its name at the console.

c**H**ain-file is selected by pressing **H**. The arrow moves to point to this line. The compiler writes code to a file with the same name as the Work file (or Main file, if specified) and the file type .CHN. This file contains the program code but no Pascal library and must be activated from another TURBO Pascal program with the *Chain* procedure (see page 231).

When the **C**md or c**H**n mode is selected, four additional lines will appear on the screen:

```
minimum cOde segment size:   XXXX paragraphs (max.YYYY)
minimum Data segment size:   XXXX paragraphs (max.YYYY)
mInimum free dynamic memory: XXXX paragraphs
mAximum free dynamic memory: XXXX paragraphs
```

*Figure 21-2: Memory Usage Menu*

The use of these commands are described in the following sections.

## Minimum Code Segment Size

The **O**-command is used to set the minimum size of the code segment for a .CMD using *Chain* or *Execute*. As discussed on page 231, *Chain* and *Execute* do not change the base addresses of the code, data, and stack segments, and a 'root' program using *Chain* or *Execute* must therefore allocate segments of sufficient size to accommodate the largest segments in any *Chained* or *Executed* program.

Consequently, when compiling a 'root' program, you must set the value of the *Minimum Code Segment Size* to at least the same value as the largest code segment size of the programs to be chained/executed from that root. The required values are obtained from the status printout terminating any compilation. The values are in hexadecimal and specify number of paragraphs, a paragraph being 16 bytes.





## Minimum Data Segment Size

The **D**-command is used to set the minimum size of the data segment for a .CMD using *Chain* or *Execute*. As discussed above, a 'root' program using these commands must allocate segments of sufficient size to accommodate the largest data of any *Chained* or *Executed* program.

Consequently, when compiling a 'root' program, you must set the value of the *Minimum Data Segment Size* to at least the same value as the largest data segment size of the programs to be chained/executed from that root. The required values are obtained from the status printout terminating any compilation. The values are in hexadecimal and specify number of paragraphs, a paragraph being 16 bytes.

## Minimum Free Dynamic Memory

This value specifies the minimum memory size required for stack and heap. The value is in hexadecimal and specifies a number of paragraphs, a paragraph being 16 bytes.

## Maximum Free Dynamic Memory

This value specifies the maximum memory size allocated for stack and heap. It must be used in programs which operate in a multi-user environment like Concurrent CP/M-86 to assure that the program does not allocate the entire free memory. The value is in hexadecimal and specifies a number of paragraphs, a paragraph being 16 bytes.

## Command Line Parameters

The **P**-command lets you enter one or more parameters which are passed to your program when running it in **M**emory mode, just as if they had been entered on the DOS command line. These parameters may be accessed through the *ParamCount* and *ParamStr* functions.

## Find Runtime Error

When you run a program compiled in memory, and a runtime error occurs, the editor is invoked, and the error is automatically pointed out. This, of course, is not possible if the program is in a .CMD file or an .CHN file. Run time errors then print out the error code and the value of the program counter at the time of the error:



ClibPDF - www.fastio.com

```
Run-time error 01, PC=1B56
Program aborted
```

*Figure 21-3: Run-time Error Message*

To find the place in the source text where the error occurred, enter the **F** command. When prompted for the address, enter the address given by the error message:

```
Enter PC: 1B56
```

*Figure 21-4: Find Run-time Error*

The place in the source text is now found and pointed out exactly as if the error had occurred while running the program in memory.

Notice that locating errors in programs using overlays can be a bit more tricky, as explained on page 156.

## Standard Identifiers

The following standard identifiers are unique to the 16-bit implementations:

| | | | |
|---|---|---|---|
| Bdos | Intr | Ofs | Seg |
| CSeg | MemW | PortW | SSeg |
| DSeg | | | |





## Chain and Execute

TURBO Pascal provides two procedures *Chain* and *Execute* which allow TURBO programs to activate other TURBO programs. The syntax of the procedure calls are:

Chain(*FilVar*)
Execute(*FilVar*)

where *FilVar* is a file variable of any type, previously assigned to a disk file with the standard procedure *Assign*. If the file exists, it is loaded into memory and executed.

The *Chain* procedure is used only to activate special TURBO Pascal .CHN files, i.e. files compiled with the c**H**n-file option selected on the **O**ptions menu (see page 190). Such a file contains only program code; no Pascal library, it uses the Pascal library already present in memory.

The *Execute* procedure is used to activate any TURBO Pascal .CMD file.

If the disk file does not exist, an I/O error occurs. This error is treated as described on page 116. When the **I** compiler directive is passive ({$I-}), program execution continues with the statement following the failed *Chain* or *Execute* statement, and the *IOresult* function must be called prior to further I/O.

Data can be transferred from the current program to the chained program either by *shared global variables* or by *absolute address variables.*

To ensure overlapping, shared global variables should be declared as the very first variables in both programs, and they must be listed in the same order in both declarations. Furthermore, both programs must be compiled to the same size of code and data segments (see pages 228 and 229). When these conditions are satisfied, the variables will be placed at the same address in memory by both programs, and as TURBO Pascal does not automatically initialize its variables, they may be shared.



ClibPDF - www.fastio.com

**Example:**
Program *MAIN.CMD:*

```
program Main;
var
  Txt:      string[80];
  CntPrg:   file;

begin
  Write('Enter any text: '); Readln(Txt);
  Assign(CntPrg, 'ChrCount.chn');
  Chain(CntPrg);
end.
```

Program *CHRCOUNT.CHN:*

```
program ChrCount;
var
  Txt:      string[80];
  NoOfChar,
  NoOfUpc,
  I:        Integer;

begin
  NoOfUpc := 0;
  NoOfChar := Length(Txt);
  for I := 1 to length(Txt) do
    if Txt[I] in ['A'..'Z'] then NoOfUpc := Succ(NoOfUpc);
  Write('No of characters in entry: ',NoOfChar);
  Writeln('. No of upper case characters: ', NoOfUpc,'.');
end.
```

If you want a TURBO program to determine whether it was invoked by eXecute or directly from the CP/M command line, you should use an **absolute** variable at address *Dseg:$80*. This is the command line length byte, and when a program is called from CP/M, it contains a value between 0 and 127. When eXecuting a program, therefore, the calling program should set this variable to something higher than 127. When you then check the variable in the called program, a value between 0 and 127 indicates that the program was called from CP/M, a higher value that it was called from another TURBO program.





Chaining and eXecuting TURBO programs does not alter the memory allocation state. The base addresses and sizes of the code, data and stack segments are not changed; *Chain* and *Execute* only replace the program code in the code segment. 'Alien' programs, therefore, cannot be initiated from a TURBO program.

It is important that the first program which executes a *Chain* statement allocates enough memory for the code, data, and stack segments to accommodate largest .CHN program. This is done by using the **O**ptions menu to change the minimum code, data and free memory sizes (see page 190).

**Note** that neither *Chain* nor *Execute* can be used in direct mode, that is, from a program run with the compiler options switch in position **M**emory (page 190).

## Overlays

During execution, the system normally expects to find its overlay files on the logged drive. The *OvrDrive* procedure may be used to change this default value.

### *OvrDrive Procedure*

**Syntax:** OvrDrive(*Drive*);

where *Drive* is an integer expression specifying a drive (0 = logged drive, 1 = A:, 2 = B:, etc.). On subsequent calls to overlay files, the files will be expected on the specified drive. Once an overlay file has been opened on one drive, future calls to the same file will look on the same drive.

**Example:**

```
program OvrTest;

overlay procedure ProcA;
begin
  Writeln('Overlay A');
end;
```



ClibPDF - www.fastio.com

```
overlay procedure ProcB;
begin
  Writeln('Overlay B');
end;

procedure Dummy;
begin
  {Dummy procedure to separate the overlays
   into two groups}
end;

overlay procedure ProcC;
begin
  Writeln('Overlay C');
end;

begin
  OvrDrive(2);
  ProcA;
  OvrDrive(0);
  ProcC;
  OvrDrive(2);
  ProcB;
end.
```

The first call to *OvrDrive* specifies overlays to be sought on the B: drive. The call to *ProcA* therefore causes the first overlay file (containing the two overlay procedures *ProcA* and *ProcB* to be opened here.

Next, the *OvrDrive(0)* statement specifies that following overlays are to be found on the logged drive. The call to *ProcC* opens the second overlay file here.

The following *ProcB* statement calls an overlay procedure in the first overlay file; and to ensure that it is sought on the B: drive, the *OvrDrive(2)* statement must be executed before the call.





# Files

## File Names

A file name in CP/M consists of one through eight letters or digits, optionally followed by a period and a file type of one through three letters or digits:

*Drive:Name.Type*

## Untyped Files

An optional second parameter on *Reset* and *ReWrite* may be used to specify the block size to be used by *BlockRead* and *BlockWrite*. For example:

```
Assign(InFile,'INDATA');
Reset(InFile,BlockSize);
```

where *BlockSize* is an integer expression.

## Text Files

The *Seek* and *Flush* procedures and the *FilePos* and *FileSize* functions are not applicable to CP/M text files.

### Buffer Size

The text file buffer size is 128 bytes by default. This is adequate for most applications, but heavily I/O-bound programs, as for example a copy program, will benefit from a larger buffer, as it will reduce disk head movement.

You are therefore given the option to specify the buffer size when declaring a text file:

```
VAR
  TextFile: Text[$1000];
```

declares a text file variable with a buffer size of 4K bytes.



ClibPDF - www.fastio.com

## Absolute Variables

Variables may be declared to reside at specific memory addresses, and are then called **absolute**. This is done by adding to the variable declaration the reserved word **absolute** followed by two *Integer* constants specifying a segment and an offset at which the variable is to be located:

```
var
  Abc: Integer  absolute $0000:$00EE;
  Def: Integer  absolute $0000:$00F0;
```

The first constant specifies the segment base address, and the second constant specifies the offset within that segment. The standard identifiers *CSeg* and *DSeg* may be used to place variables at absolute addresses within the code segment (CSeg) or the data segment (DSeg):

```
Patch: array[1..PatchSize] of byte absolute CSeg:$05F3;
```

**Absolute** may also be used to declare a variable "on top" of another variable, i.e. that a variable should start at the same address as another variable. When **absolute** is followed by the identifier of a variable or parameter, the new variable will start at the address of that variable parameter.

**Example:**

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

This declaration specifies that the variable *StrLen* should start at the same address as the variable *Str*, and as the first byte of a string variable contains the length of the string, *StrLen* will contain the length of *Str*. Notice that an **absolute** variable declaration may only specify one identifier.

Further details on space allocation for variables are found on page 246.





## Absolute Address Functions

The following functions are provided for obtaining information about program variable addresses and system pointers.

### Addr

**Syntax:** Addr(*Name*)

Returns the address in memory of the first byte of the variable with the identifier *Name*. If *Name* is an array, it may be subscribed, and if *Name* is a record, specific fields may be selected. The value returned is a 32 bit pointer consisting of a segment address and an offset.

### Ofs

**Syntax:** Ofs(*Name*)

Returns the offset in the segment of memory occupied by the first byte of the variable, procedure or function with the identifier *Name*. If *Name* is an array, it may be subscribed, and if *Name* is a record, specific fields may be selected. The value returned is an *Integer*.

### Seg

**Syntax:** Seg(*Name*)

Returns the address of the segment containing the first byte of the variable with the identifier *Name*. If *Name* is an array, it may be subscribed, and if *Name* is a record, specific fields may be selected. The value returned is an *Integer*. To obtain the segment address of a procedure or function, use the CSEG function.

### Cseg

**Syntax:** Cseg

Returns the base address of the **Code** segment. The value returned is an *Integer*.



ClibPDF - www.fastio.com

### Dseg

**Syntax:** Dseg

Returns the base address of the **Data** segment. The value returned is an *Integer.*

### Sseg

**Syntax:** Sseg

Returns the base address of the **Stack** segment. The value returned is an *Integer.*

## Predefined Arrays

TURBO Pascal offers four predefined arrays of type *Byte,* called *Mem, MemW, Port* and *PortW* which are used to access CPU memory and data ports.

### Mem Array

The predefined arrays *Mem* and *MemW* are used to access memory. Each component of the array *Mem* is a byte, and each component of the array *Wmem* is a word (two bytes, LSB first). The index must be an address specified as the segment base address and an offset separated by a colon and both of type *Integer.*

The following statement assigns the value of the byte located in segment 0000 at offset $0081 to the variable *Value*

```
Value:=Mem[0000:$0081];
```

While the following statement:

```
MemW[Seg(Var):Ofs(Var)]:=Value;
```

places the value of the *Integer* variable *Value* in the memory location occupied by the two first bytes of the variable *Var.*





### Port Array

The *Port* and *PortW* array are used to access the data ports of the 8086/88 CPU. Each element of the array represents a data port, with the index corresponding to port numbers. As data ports are selected by 16-bit addresses the index type is *Integer.* When a value is assigned to a component of *Port* or *PortW* it is output to the port specified. When a component of port is referenced in an expression, its value is input from the port specified. The components of the *Port* array are of type *Byte* and the components of *PortW* are of type *Integer.*

**Example:**
```
Port[56]:=10;
```

The use of the port array is restricted to assignment and reference in expressions only, i.e. components of *Port* and *PortW* cannot be used as variable parameters to procedures and functions. Furthermore, operations referring to the entire port array (reference without index) are not allowed.

## With Statements

*With* statements may be nested to a maximum of 9 levels.

## Pointer Related Items

### MemAvail

The standard function *MemAvail* is available to determine the available space on the heap at any given time. The result is an *Integer* specifying the number of available *paragraphs* on the heap (a *paragraph* is 16 bytes).

### Pointer Values

In very special circumstances it can be of interest to assign a specific value to a pointer variable *without using another pointer variable* or it can be of interest to obtain the actual value of a pointer variable.



ClibPDF - www.fastio.com

## Assigning a Value to a Pointer

The standard function *Ptr* can be used to assign specific values to a pointer variable. The function returns a 32 bit pointer consisting of a segment address and an offset.

**Example:**

```
Pointer:=Ptr(Cseg,$80);
```

## Obtaining The Value of a Pointer

A pointer value is represented as a 32 bit entity and the standard function *Ord* can therefore **not** be used to obtain its value. Instead the functions *Ofs* and *Seg* must be used.

The following statement obtains the value of the pointer *P* (which is a segment address and an offset):

```
SegmentPart:=Seg(P^);
OffsetPart:=Ofs(P^);
```

# Function Calls

For the purpose of calling the CP/M-86 BDOS, TURBO Pascal introduces a procedure *Bdos*, which has a record as parameter.

Details on BDOS and BIOS routines are found in the *CP/M-86 Operating System Manual* published by Digital Research.

The parameter to *Bdos* must be of the type:

```
record
  AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
end;
```

Before TURBO calls the BDOS, the registers AX, BX, CX, DX, BP, SI, DI, DS, and ES are loaded with the values specified in the record parameter. When the BDOS has finished operation the *Bdos* procedure will restore the registers to the record thus making any results from the BDOS available.





# User Written I/O Drivers

For some applications it is practical for a programmer to define his own input and output drivers, i.e. routines which perform input and output of characters to and from an external device. The following drivers are part of the TURBO environment, and used by the standard I/O drivers (although they are not available as standard procedures or functions):

**function** *ConSt*: boolean; { 6 }
**function** *ConIn*: Char; { 6 }
**procedure** *ConOut*(Ch: Char); { 6 }
**procedure** *LstOut*(Ch: Char); { 5 }
**procedure** *AuxOut*(Ch: Char); { 4 }
**function** *AuxIn*: Char; { 3 }
**procedure** *UsrOut*(Ch: Char); { 6 }
**function** *UsrIn*: Char; { 6 }

The *ConSt* routine is called by the function *KeyPressed*, the *ConIn* and *ConOut* routines are used by the CON:, TRM:, and KBD: devices, the *LstOut* routine is used by the LST: device, the *AuxOut* and *AuxIn* routines are used by the AUX: device, and the *UsrOut* and *UsrIn* routines are used by the USR: device.

By default, these drivers are assigned to the BDOS functions as showed in curly braces in the above listing of drivers.

This, however, may be changed by the programmer by assigning the address of a self-defined driver procedure or a driver function to one of the following standard variables:

| Variable | Contains the address of the |
|---|---|
| *ConStPtr* | *ConSt* function |
| *ConInPtr* | *ConIn* function |
| *ConOutPtr* | *ConOut* procedure |
| *LstOutPtr* | *LstOut* procedure |
| *AuxOutPtr* | *AuxOut* procedure |
| *AuxInPtr* | *AuxIn* function |
| *UsrOutPtr* | *UsrOut* procedure |
| *UsrInPtr* | *UsrIn* function |



ClibPDF - www.fastio.com

A user defined driver procedure or driver function must match the definitions given above, i.e. a *ConSt* driver must be a boolean function, a *ConIn* driver must be a char function, etc.

## External Subprograms

The reserved word **external** is used to declare external procedures and functions, typically procedures and functions written in machine code.

The reserved word **external** must be followed by a string constant specifying the name of a file in which executable machine code for the external procedure or function must reside.

During compilation of a program containing external functions or procedures the associated files are loaded and placed in the object code. Since it is impossible to know beforehand exactly *where* in the object code the external code will be placed this code **must** be relocatable, and no references must be made to the data segment. Furthermore the external code must save the registers BP, CS, DS and SS and restore these before executing the RET instruction.

An external subprogram has no *block*, i.e. no declaration part and no statement part. Only the subprogram heading is specified, immediately followed by the reserved word **external** and a filename specifying where to find the executable code for the subprogram.

The type of the filename is *.CMD*. Only the code segment of a .CMD file is loaded.

**Example:**

```
procedure DiskReset; external 'DSKRESET';
function IOstatus: boolean; external 'IOSTAT';
```

Parameters may be passed to external subprograms, and the syntax is exactly the same as that of calls to ordinary procedures and functions:

```
procedure Plot(X,Y: Integer); external 'PLOT';
procedure QuickSort(var List: PartNo); external 'QS';
```

External subprograms and parameter passing is discussed further on page 252.





## In-line Machine Code

TURBO Pascal features the **inline** statements as a very convenient way of inserting machine code instructions directly into the program text. An inline statement consists of the reserved word **inline** followed by one or more *code elements* separated by slashes and enclosed in parentheses.

A code element is built from one or more data elements, separated by plus (+) or minus (−) signs. A data element is either an integer constant, a variable identifier, a procedure identifier, a function identifier, or a location counter reference. A location counter reference is written as an asterisk (*).

**Example:**

```
inline (10/$2345/count+1/sort-*+2);
```

Each code element generates one byte or one word (two bytes) of code. The value of the byte or the word is calculated by adding or subtracting the values of the data elements according to the signs that separate them. The value of a variable identifier is the address (or offset) of the variable. The value of a procedure or function identifier is the address (or offset) of the procedure or function. The value of a location counter reference is the address (or offset) of the location counter, i.e. the address at which to generate the next byte of code.

A code element will generate one byte of code if it consists of integer constants only, and if its value is within the 8-bit range (0..255). If the value is outside the 8-bit range, or if the code element refers to variable, procedure, or function identifiers, or if the code element contains a location counter reference, one word of code is generated (least significant byte first).

The '<' and '>' characters may be used to override the automatic size selection described above. If a code element starts with a '<' character, only the least significant byte of the value is coded, even if it is a 16-bit value. If a code element starts with a '>' character, a word is always coded, even though the most significant byte is zero.

**Example:**

```
inline (<$1234/>$44);
```

This **inline** statement generates three bytes of code: $34, $44, $00.



ClibPDF - www.fastio.com

The value of a variable identifier use in a **inline** statement is the offset address of the variable within its base segment. The base segment of global variables (i.e. variables declared in the main program block) is the data segment, which is accessible through the DS register. The base segment of local variables (i.e. variables declared within the current subprogram) is the stack segment, and in this case the variable offset is relative to the BP (base page) register, the use of which automatically causes the stack segment to be selected. The base segment of typed constants is the code segment, which is accessible through the CS register. **inline** statements should not attempt to access variables that are not declared in the main program nor in the current subprogram.

The following example of an inline statement generates machine code that will convert all characters in its string argument to upper case.

```
procedure UpperCase(var Strg: Str);
{Str is type String[255]}
begin
  inline
    ($C4/$BE/Strg/      {      LES   DI,Strg[BP]          }
     $26/$8A/$0D/       {      MOV   CL,ES:[DI]           }
     $FE/$C1/           {      INC   CL                   }
     $FE/$C9/           { L1:  DEC   CL                   }
     $74/$13/           {      JZ    L2                   }
     $47/               {      INC   DI                   }
     $26/$80/$3D/$61/   {      CMP   ES:BYTE PTR [DI],'a'}
     $72/$F5/           {      JB    L1                   }
     $26/$80/$3D/$7A/   {      CMP   ES:BYTE PTR [DI],'z'}
     $77/$EF/           {      JA    L1                   }
     $26/$80/$2D/$20/   {      SUB   ES:BYTE PTR [DI],20H}
     $EB/$E9);          {      JMP   SHORT L1             }
                        { L2:                             }
end;
```

**Inline** statements may be freely mixed with other statements throughout the statement part of a block, and **inline** statements may use all CPU registers. **Note**, however, that the contents of the registers BP, SP, DS, and SS must be the same on exit as on entry.





## Interrupt Handling

A TURBO Pascal interrupt routine must manually preserve registers AX, BX, CX, DX, SI, DI, DS and ES. This is done by placing the following inline statement as the first statement of the procedure:

```
inline ($50/$53/$51/$52/$56/$57/$1E/$06/$FB);
```

The last byte ($FB) is an STI instruction which enables further interrupts - it may or may not be required. The following inline statement must be the last statement in the procedure:

```
inline ($07/$1F/$5F/$5E/$5A/$59/$5B/$58/$8B/$E5/$5D/$CF);
```

This restores the registers and reloads the stack pointer (SP) and the base page register (BP). The last byte ($CF) is an IRET instruction which overrides the RET instruction generated by the compiler.

An interrupt service procedure must not employ any I/O operations using the standard procedures and functions of TURBO Pascal, as the BDOS is not re-entrant. The programmer must initialize the interrupt vector used to activate the interrupt service routine.

### Intr procedure

**Syntax:** Intr(*InterruptNo, Result*)

This procedure initializes the registers and flags as specified in the parameter *Result* which must be of type:

```
Result = record
           AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
         end;
```

It then makes the software interrupt given by the parameter *interruptNo* which must be an *Integer* constant. When the interrupt service routine returns control to your program, *Result* will contain any values returned from the service routine.



ClibPDF - www.fastio.com

Note that the data segment register DS, used to access global variables, will not have the correct value when the interrupt service routine is entered. Therefore, global variables cannot be directly accessed. *Typed constants*, however, are available, as they are stored in the code segment. The way to access global variables in the interrupt service routine is therefore to store the value of *Dseg* in a typed constant in the main program. This typed constant can then be accessed by the interrupt handler and used to set its DS register.

## Internal Data Formats

In the following descriptions, the symbol @ denotes the offset of the first byte occupied by a variable of the given type within its segment. The segment base address can be determined by using the standard function *Seg*.

*Global* and *local variables*, and *typed constants* occupy different segments as follows:

**Global variables** reside in the data segment and the offset is relative to the DS register.

**Local variables** reside in the stack segment and the offset is relative to the BP register.

**Typed constants** reside in the code segment and the offset is relative to the CS register.

All variables are contained within their base segment.

### Basic Data Types

The basic data types may be grouped into structures (arrays, records, and disk files), but this structuring will not affect their internal formats.





### Scalars

The following scalars are all stored in a single byte: *Integer* subranges with both bounds in the range 0..255, booleans, chars, and declared scalars with less than 256 possible values. This byte contains the ordinal value of the variable.

The following scalars are all stored in two bytes: *Integer*s, *Integer* subranges with one or both bounds not within the range 0..255, and declared scalars with more than 256 possible values. These bytes contain a 2's complement 16-bit value with the least significant byte stored first.

### Reals

Reals occupy 6 bytes, giving a floating point value with a 40-bit mantissa and an 8-bit 2's exponent. The exponent is stored in the first byte and the mantissa in the next five bytes with the least significant byte first:

| | |
|---|---|
| @ | Exponent |
| @+1 | LSB of mantissa |
| : | |
| @+5 | MSB of mantissa |

The exponent uses binary format with an offset of $80. Hence, an exponent of $84 indicates that the value of the mantissa is to be multiplied by 2^($84-$80) = 2^4 = 16. If the exponent is zero, the floating point value is considered to be zero.

The value of the mantissa is obtained by dividing the 40-bit unsigned integer by 2^40. The mantissa is always normalized, i.e. the most significant bit (bit 7 of the fifth byte) should be interpreted as a 1. The sign of the mantissa is stored in this bit, however, a 1 indicating that the number is negative, and a 0 indicating that the number is positive.



ClibPDF - www.fastio.com

## Strings

A string occupies as many bytes as its maximum length plus one. The first byte contains the current length of the string. The following bytes contains the string with the first character stored at the lowest address. In the table shown below, **L** denotes the current length of the string, and *Max* denotes the maximum length:

| | |
|---|---|
| @ | Current length (*L*) |
| @+1 | First character |
| @+2 | Second character |
| : | |
| @+*L* | Last character |
| @+*L*+1 | Unused |
| : | |
| @+*Max* | Unused |

## Sets

An element in a *Set* occupies one bit, and as the maximum number of elements in a set is 256, a set variable will never occupy more than 32 bytes (256/8).

If a set contains less than 256 elements, some of the bits are bound to be zero at all times and need therefore not be stored. In terms of memory efficiency, the best way to store a set variable of a given type would then be to "cut off" all insignificant bits, and rotate the remaining bits so that the first element of the set would occupy the first bit of the first byte. Such rotate operations, however, are quite slow, and TURBO therefore employs a compromise: Only bytes which are statically zero (i.e. bytes of which no bits are used) are not stored. This method of compression is very fast and in most cases as memory efficient as the rotation method.

The number of bytes occupied by a set variable is calculated as (*Max* **div** 8) − (*Min* **div** 8) + 1, where *Max* and *Min* are the upper and lower bounds of the base type of that set. The memory address of a specific element *E* is:

*MemAddress* = @ + (*E* **div** 8) − (*Min* **div** 8)

and the bit address within the byte at MemAddress is:





*BitAddress* = *E* **mod** 8

where *E* denotes the ordinal value of the element.

## Pointers

A pointer consists of four bytes containing a segment base address and an offset. The two least significant bytes contains the offset and the two most significant bytes the base address. Both are stored in memory using byte reversed format, i.e. the least significant byte is stored first. The value **nil** corresponds to two zero words.

## Data Structures

Data structures are built from the basic data types using various structuring methods. Three different structuring methods exist: Arrays, records, and disk files. The structuring of data does not in any way affect the internal formats of the basic data types.

## Arrays

The components with the lowest index values are stored at the lowest memory address. A multi-dimensional array is stored with the rightmost dimension increasing first, e.g. given the array

```
Board: array[1..8,1..8] of Square
```

you have the following memory layout of its components:

```
lowest address:  Board[1,1]
                 Board[1,2]
                 :
                 Board[1,8]
                 Board[2,1]
                 Board[2,2]
                 :
                 :
Highest address: Board[8,8]
```



ClibPDF - www.fastio.com

## Records

The first field of a record is stored at the lowest memory address. If the record contains no variant parts, the length is given by the sum of the lengths of the individual fields. If a record contains a variant, the total number of bytes occupied by the record is given by the length of the fixed part plus the length of largest of its variant parts. Each variant starts at the same memory address.

## Disk Files

Disk files are different from other data structures in that data is not stored in internal memory but in a file on an external device. A disk file is controlled through a file interface block (FIB).

### File Interface Blocks

The table below shows the format of a FIB:

| | |
|---|---|
| @ + 0 | Flags byte. |
| @ + 1 | Character buffer. |
| @ + 2 | Number of records (LSB) or buffer offset (LSB). |
| @ + 3 | Number of records (MSB) or buffer offset (MSB). |
| @ + 4 | Record length (LSB) or buffer size (LSB). |
| @ + 5 | Record length (MSB) or buffer size (MSB). |
| @ + 6 | Buffer pointer (LSB). |
| @ + 7 | Buffer pointer (MSB). |
| @ + 8 | Current record (LSB) or buffer end (LSB). |
| @ + 9 | Current record (MSB) or buffer end (LSB). |
| @ + 10 | Unused. |
| @ + 11 | Unused. |
| @ + 12 | First byte of CP/M FCB. |
| : | |
| @ + 47 | Last byte of CP/M FCB. |
| @ + 48 | First byte of sector buffer. |
| : | |
| @ + 175 | Last byte of sector buffer. |





The format of the flags byte at @ + 0 is:

| | |
|---|---|
| Bit 0..3 | File type. |
| Bit 4 | Read semaphore. |
| Bit 5 | Write semaphore or pre-read character flag. |
| Bit 6 | Output flag. |
| Bit 7 | Input flag. |

File type 0 denotes a disk file, and 1 through 5 denote the TURBO Pascal logical I/O devices (CON:, KBD:, LST:, AUX:, and USR:). For typed files, bit 4 is set if the contents of the sector buffer is undefined, and bit 5 is set if data has been written to the sector buffer. For textfiles, bit 5 is set if the character buffer contains a pre-read character. Bit 6 is set if output is allowed, and bit 7 is set if input is allowed.

For typed and untyped files, the four words from @ + 2 to @ + 9 store the number of records in the file, the record length in bytes, the sector buffer pointer, and the current record number. For typed files, the sector buffer pointer stores an offset (0..127) in the sector buffer at @ + 48. The FIB of an untyped file has no sector buffer, and so the sector buffer pointer is not used.

For text files, the four words from @ + 2 to @ + 9 store the offset address of the buffer, its size, the offset of the next character to read or write, and the offset of the first byte after the buffer. The buffer always resides in the same segment as the FIB, usually starting at @ + 48. The size of a textfile FIB may be larger than indicated, depending on the size of the buffer. When a textfile is assigned to a logical device, only the flags byte and the character buffer are used.

### Random Access Files

A random access file consists of a sequence of records, all of the same length and same internal format. To optimize file storage capacity, the records of a file are totally contiguous. The first four bytes of the first sector of a file contains the number of records in the file and the length of each record in bytes. The first record of the file is stored starting at the fourth byte.



ClibPDF - www.fastio.com

| | |
|---|---|
| sector 0, byte 0: | Number of records (LSB) |
| sector 0, byte 1: | Number of records (MSB) |
| sector 0, byte 2: | Record length (LSB) |
| sector 0, byte 3: | Record length (MSB) |

**Text Files**

The basic components of a text file are characters, but a text file is furthermore divided into *lines*. Each line consists of any number of characters ended by a CR/LF sequence (ASCII $0D/ $0A). The file is terminated by a Ctrl-Z (ASCII $1B).

## Parameters

Parameters are transferred to procedures and functions via the stack which is addressed through SS:SP.

On entry to an **external** subroutine, the top of the stack always contains the return address within the code segment (a word). The parameters, if any, are located below the return address, i.e. at higher addresses on the stack.

If an external function has the following subprogram header:

```
function Magic(var R: Real; S: string5): Integer;
```

then the stack upon entry to *Magic* would have the following contents:

```
< Function result            >
< Segment base address of R  >
< Offset address of R        >
< First character of S       >
:
< Last character of S        >
< Length of S                >
< Return address             >  SP
```

An external subroutine should save the Base Page register (BP) and then copy the Stack Pointer SP into the Base Page register in order to be able to refer to parameters. Furthermore the subroutine should reserve space on the stack for local workarea. This can be obtained by the following instructions:





```
PUSH  BP
MOV   BP,SP
SUB   SP,WORKAREA
```

The last instruction will have the effect of adding the following to the stack:

```
< Return address              >  BP
< The saved BP register       >
< First byte of local workarea >
:
< Last byte of local work area >  SP
```

Parameters are accessed via the BP register.

The following instruction will load length of the string into the AL register:

```
MOV  AL,[BP+4]
```

Before executing a RET instruction the subprogram must reset the Stack Pointer and Base Page register to their original values. When executing the RET the parameters may be removed by giving RET a parameter specifying how many bytes to remove. The following instructions should therefore be used when exiting from a subprogram:

```
MOV  SP,BP
POP  BP
RET  NoOfBytesToRemove
```

### Variable Parameters

With a variable (**var**) parameter, two words are transferred on the stack giving the base address and offset of the first byte occupied by the actual parameter.

### Value Parameters

With value parameters, the data transferred on the stack depends upon the type of the parameter as described in the following sections.



ClibPDF - www.fastio.com

### Scalars

*Integers, Booleans, Chars* and declared scalars (i.e. all scalars except *Reals*) are transferred on the stack as a word. If the variable occupies only one byte when it is stored, the most significant byte of the parameter is zero.

### Reals

A real is transferred on the stack using six bytes.

### Strings

When a string is at the top of the stack, the topmost byte contains the length of the string followed by the characters of the string.

### Sets

A set always occupies 32 bytes on the stack (set compression only applies to the loading and storing of sets).

### Pointers

A pointer value is transferred on the stack as two words containing the base address and offset of a dynamic variable. The value NIL corresponds to two zero words.

### Arrays and Records

Even when used as value parameters, *Array* and *Record* parameters are not actually transferred on the stack. Instead, two words containing the base address and offset of the first byte of the parameter are transferred. It is then the responsibility of the subroutine to use this information to make a local copy of the variable.





## Function Results

User written **external** functions must remove all parameters and the function result from the stack when they return.

User written **external** functions must return their results exactly as specified in the following:

Values of scalar types, except *Reals*, must be returned in the AX register. If the result is only one byte then AH should be set to zero. Boolean functions must return the function value by setting the Z flag (Z = *False*, NZ = *True*).

*Reals* must be returned on the stack with the exponent at the lowest address. This is done by not removing the function result variable when returning.

**Sets** must be returned on the top of the stack according to the format described on page 254. On exit SP must point at the byte containing the string length.

Pointer values must be returned in the DX:AX.

## The Heap and The Stacks

During execution of TURBO Pascal program the following segments are allocated for the program:

a Code Segment,
a Data Segment, and
a Stack Segment

Two stack-like structures are maintained during execution of a program: the *heap* and the *stack*.

The heap is used to store dynamic variables, and is controlled with the standard procedures *New, Mark,* and *Release*. At the beginning of a program, the heap pointer *HeapPtr* is set to low memory in the stack segment and the heap grows upwards towards the stack. The predefined variable *HeapPtr* contains the value of the heap pointer and allows the programmer to control the position of the heap.



ClibPDF - www.fastio.com

The stack is used to store local variables, intermediate results during evaluation of expressions and to transfer parameters to procedures and functions. At the beginning of a program, the stack pointer is set to the address of the top of the stack segment.

On each call to the procedure *New* and on entering a procedure or function, the system checks for collision between the heap and the recursion stack. If a collision has occurred, an execution error results, unless the **K** compiler directive is passive ({ $K-}).

## Memory Management

When a TURBO program is executed, three segments are allocated for the program: A code segment, a data segment, and a stack segment.

Code segment (CS is the code segment register):

```
CS:0000 - CS:EOFR    Run-time library code.
CS:EOFR - CS:EOFP    Program code.
CS:EOFP - CS:EOFC    Unused.
```

Data segment (DS is the data segment register):

```
DS:0000 - DS:00FF    CP/M-86 base page.
DS:0100 - DS:EOFW    Run-time library workspace.
DS:EOFW - DS:EOFM    Main program block variables.
DS:EOFM - DS:EOFD    Unused.
```

The unused areas between (CS:EOFP-CS:EOFC and DS:EOFM-DS:EOFD) are allocated only if a minimum cOde segment size larger than the required size is specified at compilation. The sizes of the code and data segments never exceed 64K bytes each.

The stack segment is slightly more complicated, as it may be larger than 64K bytes. On entry to the program the stack segment register (SS) and the stack pointer (SP) is loaded so that SS:SP points at the very last byte available in the entire segment. During execution of the program SS is never changed but SP may move downwards until it reaches the bottom of the segment, or 0 (corresponding to 64K bytes of stack) if the stack segment is larger than 64K bytes.





The heap grows from low memory in the stack segment towards the actual stack residing in high memory. Each time a variable is allocated on the heap, the heap pointer (which is a double word variable maintained by the TURBO run-time system) is moved upwards, and then normalized, so that the offset address is always between $0000 and $000F. Therefore, the maximum size of a single variable that can be allocated on the heap is 65521 bytes (corresponding to $10000 less $000F). The total size of all variables allocated on the heap is however only limited by the amount of memory available.

The heap pointer is available to the programmer through the *HeapPtr* standard identifier. *HeapPtr* is a typeless pointer which is compatible with all pointer types. Assignments to *HeapPtr* should be exercised only with extreme care.



ClibPDF - www.fastio.com

**Notes:**



# Chapter 22
# CP/M-80

This chapter describes features of TURBO Pascal specific to the 8-bit CP/M-80 implementation. It presents two kinds of information:

1) Things you should know to make efficient use of TURBO Pascal. Pages 259 through 272.

2) The rest of the chapter describes things which are only of interest to experienced programmers, such as machine language routines, technical aspects of the compiler, etc.

## eXecute Command

You will find an additional command on the main TURBO menu in the CP/M-80 version: **eXecute**. It lets you run other programs from within TURBO Pascal, for example copying programs, word processors - in fact anything that you can run from your operating system. When entering **X**, you are prompted:

```
Command: ■
```

You may now enter the name of any program which will then load and run normally. Upon exit from the program, control is re-transferred to TURBO Pascal, and you return to the TURBO prompt > .

## compiler Options

The **O** command selects the following menu on which you may view and change some default values of the compiler. It also provides a helpful function to find runtime errors in programs compiled into object code files.



ClibPDF - www.fastio.com

```
compile -> Memory
           Com-file
           cHn-file

command line Parameter:

Find run-time error  Quit
```

*Figure 22-1: Options Menu*

## Memory / Com file / cHn-file

The three commands **M**, **C**, and **H** select the compiler mode, i.e. where to put the code which results from the compilation.

**M**emory is the default mode. When active, code is produced in memory and resides there ready to be activated by a **R**un command.

**C**om-file is selected by pressing **C**. The arrow moves to point to this line. When active, code is written to a file with the same name as the Work file (or Main file, if specified) and the file type .COM. This file contains the program code and Pascal runtime library, and may be activated by typing its name at the console. Programs compiled this way may be larger than programs compiled in memory, as the program code itself does not take up memory during compilation, and as program code starts at a lower address.

c**H**ain-file is selected by pressing **H**. The arrow moves to point to this line. When active, code is written to a file with the same name as the Work file (or Main file, if specified) and the file type .CHN. This file contains the program code but no Pascal library and must be activated from another TURBO Pascal program with the *Chain* procedure (see page 263).

When **C**om or c**H**n mode is selected, the menu is expanded with the following two lines:





```
Start address: XXXX (min YYYY)
End   address: XXXX (max YYYY)
```

*Figure 22-2: Start and End Addresses*

## Start Address

The Start address specifies the address (in hexadecimal) of the first byte of the code. This is normally the end address of the Pascal library plus one, but may be changed to a higher address if you want to set space aside e.g. for absolute variables to be shared by a series of chained programs.

When you enter an **S**, you are prompted to enter a new Start address. If you just hit <RETURN>, the minimum value is assumed. Don't set the Start address to anything less than the minimum value, as the code will then overwrite part of the Pascal library.

## End Address

The End address specifies the highest address available to the program (in hexadecimal). The value in parentheses indicate the top of the TPA on your computer, i.e. BDOS minus one. The default setting is 700 to 1000 bytes less to allow space for the loader which resides just below BDOS when executing programs from TURBO.

If compiled programs are to run in a different environment, the **E**nd address may be changed to suit the TPA size of that system. If you anticipate your programs to run on a range of different computers, it will be wise to set this value relatively low, e.g. C100 (48K), or even A100 (40K) if the program is to run under MP/M.



ClibPDF - www.fastio.com

When you enter an **E**, you are prompted to enter a End address. If you just hit <RETURN>, the default value is assumed (i.e. top of TPA less 700 to 1000 bytes). If you set the **E**nd address higher than this, the resulting programs cannot be executed from TURBO, as they will overwrite the TURBO loader; and if you set it higher than the TPA top, the resulting programs will overwrite part of BDOS if run on your machine.

## *Command Line Parameters*

The **P**-command lets you enter one or more parameters which are passed to your program when running it in **M**emory mode, just as if they had been entered on the DOS command line. These parameters may be accessed through the *ParamCount* and *ParamStr* functions.

## *Find Runtime Error*

When you run a program compiled in memory, and a runtime error occurs, the editor is invoked, and the error is automatically pointed out. This, of course, is not possible if the program is in a .COM file or an .CHN file. Run time errors then print out the error code and the value of the program counter at the time of the error, e.g.:

```
Run-time error 01, PC=1B56
Program aborted
```

*Figure 22-3: Run-time Error Message*

To find the place in the source text where the error occurred, enter the **F** command on the **O**ptions menu. When prompted for the address, enter the address given by the error message:

```
Enter PC: 1B56
```

*Figure 22-4: Find Run-time Error*

The place in the source text is now found and pointed out exactly as if the error had occurred while running the program in memory.





## Standard Identifiers

The following standard identifiers are unique to the CP/M-80 implementation:

| | | |
|---|---|---|
| *Bios* | *Bdos* | *RecurPtr* |
| *BiosHL* | *BdosHL* | *StackPtr* |

## Chain and Execute

TURBO Pascal provides two standard procedures: *Chain* and *Execute* which allow you to activate other programs from a TURBO program. The syntax of these procedure calls is:

```
Chain(FilVar)
Execute(FilVar)
```

where *FilVar* is a file variable of any type, previously assigned to a disk file with the standard procedure *Assign.* If the file exists, it is loaded into memory and executed.

The *Chain* procedure is used only to activate special TURBO Pascal .CHN files, i.e. files compiled with the c**H**n-file option selected on the **O**ptions menu (see page 260). Such a file contains only program code; no Pascal library. It is loaded into memory and executed at the start address of the current program, i.e. the address specified when the current program was compiled. It then uses the Pascal library already present in memory. Thus, the current program and the chained program must use the same start address.

The *Execute* procedure may be used to execute any .COM file, i.e. any file containing executable code. This could be a file created by TURBO Pascal with the **C**om-option selected on the **O**ptions menu (see page 260). The file is loaded and executed at address $100, as specified by the CP/M standard.

If the disk file does not exist, an I/O error occurs. This error is treated as described on page 116. If the **I** compiler directive is passive ({ $I-}), program execution continues with the statement following the failed *Chain* or *Execute* statement, and the *IOresult* function must be called prior to further I/O.



ClibPDF - www.fastio.com

Data can be transferred from the current program to the chained program either by *shared global variables* or by *absolute address variables*.

To ensure overlapping, shared global variables should be declared as the very first variables in both programs, and they must be listed in the same order in both declarations. Furthermore, both programs must be compiled to the same memory size (see page 261). When these conditions are satisfied, the variables will be placed at the same address in memory by both programs, and as TURBO Pascal does not automatically initialize its variables, they may be shared.

**Example:**
Program *MAIN.COM:*

```
program Main;
var
  Txt:      string[80];
  CntPrg:   file;
begin
  Write('Enter any text: '); Readln(Txt);
  Assign(CntPrg, 'ChrCount.chn');
  Chain(CntPrg);
end.
```

Program *CHRCOUNT.CHN:*

```
program ChrCount;
var
  Txt:      string[80];
  NoOfChar,
  NoOfUpc,
  I:        Integer;
begin
  NoOfUpc := 0;
  NoOfChar := Length(Txt);
  for I := 1 to length(Txt) do
    if Txt[I] in ['A'..'Z'] then NoOfUpc := Succ(NoOfUpc);
  Write('No of characters in entry: ',NoOfChar);
  Writeln('. No of upper case characters: ', NoOfUpc,'.');
end.
```





If you want a TURBO program to determine whether it was invoked by eXecute or directly from the DOS command line, you should use an **absolute** variable at address *$80*. This is the command line length byte, and when a program is called from CP/M, it contains a value between 0 and 127. When eXecuting a program, therefore, the calling program should set this variable to something higher than 127. When you then check the variable in the called program, a value between 0 and 127 indicates that the program was called from CP/M, a higher value that it was called from another TURBO program.

**Note** that neither *Chain* nor *Execute* can be used in direct mode, i.e. from a program run with the compiler options switch in position **M**emory (page 260).

## Overlays

During execution, the system normally expects to find its overlay files on the logged drive. The *OvrDrive* procedure may be used to change this default value.

### *OvrDrive Procedure*

**Syntax:** OvrDrive(*Drive*)

where *Drive* is an integer expression specifying a drive (0 = logged drive, 1 = A:, 2 = B:, etc.). On subsequent calls to overlay files, the files will be expected on the specified drive. Once an overlay file has been opened on one drive, future calls to the same file will look on the same drive.

**Example :**

```
program OvrTest;

overlay procedure ProcA;
begin
  Writeln('Overlay A');
end;

overlay procedure ProcB;
begin
  Writeln('Overlay B');
end;
```



ClibPDF - www.fastio.com

```
procedure Dummy;
begin
  {Dummy procedure to separate the overlays
   into two groups}
end;

overlay procedure ProcC;
begin
  Writeln('Overlay C');
end;

begin
  OvrDrive(2);
  ProcA;
  OvrDrive(0);
  ProcC;
  OvrDrive(2);
  ProcB;
end.
```

The first call to *OvrDrive* specifies overlays to be sought on the B: drive. The call to *ProcA* therefore causes the first overlay file (containing the two overlay procedures *ProcA* and *ProcB* to be opened here.

Next, the *OvrDrive(0)* statement specifies that following overlays are to be found on the logged drive. The call to *ProcC* opens the second overlay file here.

The following *ProcB* statement calls an overlay procedure in the first overlay file; and to ensure that it is sought on the B: drive, the *OvrDrive(2)* statement must be executed before the call.





# Files

## File Names

A file name in CP/M consists of one through eight letters or digits, optionally followed by a period and a file type of one through three letters or digits:

*Drive:Name.Type*

## Text Files

The *Seek* and *Flush* procedures and the *FilePos* and *FileSize* functions are not applicable to CP/M text files.

# Absolute Variables

Variables may be declared to reside at specific memory addresses, and are then called **absolute**. This is done by adding the reserved word **absolute** and an address expressed by an integer constant to the variable declaration.

**Example:**

```
var
IObyte: Byte absolute $0003;
CmdLine: string[127] absolute $80;
```

**Absolute** may also be used to declare a variable "on top" of another variable, i.e. that a variable should start at the same address as another variable. When **absolute** is followed by the variable (or parameter) identifier, the new variable will start at the address of that variable (or parameter).

**Example:**

```
var
Str: string[32];
StrLen: Byte absolute Str;
```

The above declaration specifies that the variable *StrLen* should start at the same address as the variable *Str*, and since the first byte of a string variable gives the length of the string, *StrLen* will contain the length of *Str*. Notice that only one identifier may be specified in an **absolute** declaration, i.e. the construct:



ClibPDF - www.fastio.com

```
Ident1, Ident2: Integer absolute $8000
```

is **illegal**. Further details on space allocation for variables are given on pages 278 and 288.

## Addr Function

**Syntax:** Addr(*name*);

Returns the address in memory of the first byte of the type, variable, procedure, or function with the identifier *name*. If *name* is an array, it may be subscribed, and if *name* is a record, specific fields may be selected. The value returned is of type *Integer*.

## Predefined Arrays

TURBO Pascal offers two predefined arrays of type *Byte*, called *Mem* and *Port*, which are used to directly access CPU memory and data ports.

### *Mem Array*

The predeclared array *Mem* is used to access memory. Each component of the array is a *Byte*, and indexes correspond to addresses in memory. The index type is *Integer*. When a value is assigned to a component of *Mem*, it is stored at the address given by the index expression. When the *Mem* array is used in an expression, the byte at the address specified by the index is used.

**Examples:**

```
Mem[WsCursor] := 2;
Mem[WsCursor+1] := $1B;
Mem[WsCursor+2] := Ord(' ');
IObyte := Mem[3];
Mem[Addr+Offset] := Mem[Addr];
```





### *Port Array*

The *Port* array is used to access the data ports of the Z-80 CPU. Each element of the array represents a data port with indexes corresponding to port numbers. As data ports are selected by 8-bit addresses, the index type is *Byte*. When a value is assigned to a component of *Port*, it is output to the port specified. When a component of *Port* is referenced in an expression, its value is input from the port specified.

The use of the port array is restricted to assignment and reference in expressions only, i.e. components of *Port* cannot function as variable parameters to procedures and functions. Furthermore, operations referring to the entire *Port* array (reference without index) are not allowed.

## Array Subscript Optimization

The **X** compiler directive allows the programmer to select whether array subscription should be optimized with regard to execution speed or to code size. The default mode is active, i.e. { $X + }, which causes execution speed optimization. When passive, i.e. { $X-}, the code size is minimized.

## With Statements

The default 'depth' of nesting of *With* statements is 2, but the **W** directive may be used to change this value to between 0 and 9. For each block, *With* statements require two bytes of storage for each nesting level allowed. Keeping the nesting to a minimum may thus greatly affect the size of the data area in programs with many subprograms.



ClibPDF - www.fastio.com

## Pointer Related Items

### *MemAvail*

The standard function *MemAvail* is available to determine the available space on the heap at any given time. The result is an *Integer*, and if more than 32767 bytes is available, *MemAvail* returns a negative number. The correct number of free bytes is then calculated as 65536.0 + *MemAvail*. Notice the use of a real constant to generate a *Real* result, as the result is greater than *GMaxInt*. Memory management is discussed in further detail on page 288.

### *Pointers and Integers*

The standard functions *Ord* and *Ptr* provide direct control of the address contained in a pointer. *Ord* returns the address contained in its pointer argument as an *Integer*, and *Ptr* converts its *Integer* argument into a pointer which is compatible with all pointer types.

These functions are extremely valuable in the hands of an experienced programmer as they allow a pointer to point to anywhere in memory. If used carelessly, however, they are very dangerous, as a dynamic variable may be made to overwrite other variables, or even program code.





## CP/M Function Calls

For the purpose of calling CP/M BDOS and BIOS routines, TURBO Pascal introduces two standard procedures: *Bdos* and *Bios*, and four standard functions: *Bdos, BdosHL, Bios,* and *BiosHL*.

Details on BDOS and BIOS routines are found in the *CP/M Operating System Manual* published by Digital Research.

### *Bdos procedure and function*

**Syntax:** Bdos(*Func* {, *Param* } );

The *Bdos* **procedure** is used to invoke CP/M BDOS routines. *Func* and *Param* are integer expressions. *Func* denotes the number of the called routine and is loaded into the C register. *Param* is optional and denotes a parameter which is loaded into the DE register pair. A call to address 5 then invokes the BDOS.

The *Bdos* **function** is called like the procedure and returns an *Integer* result which is the value returned by the BDOS in the A register.

### *BdosHL function*

**Syntax:** BdosHL(*Func* {, *Param* } );

This function is exactly similar to the *Bdos* function above, except that the result is the value returned in the HL register pair.



ClibPDF - www.fastio.com

## Bios procedure and function

**Syntax:** Bios(*Func* {, *Param* } );

The *Bios* **procedure** is used to invoke BIOS routines. *Func* and *Param* are integer expressions. *Func* denotes the number of the called routine, with 0 meaning the WBOOT routine, 1 the CONST routine, etc. I.e. the address of the called routine is *Func* * 3 plus the WBOOT address contained in addresses 1 and 2. *Param* is optional and denotes a parameter which is loaded into the BC register pair prior to the call.

The *Bios* **function** is called like the procedure and returns an integer result which is the value returned by the BIOS in the A register.

## BiosHL function

**Syntax:** BiosHL(*Func* {, *Param* } );

This function is exactly similar to the *Bios* function above, except that the result is the value returned in the HL register pair.

# User Written I/O Drivers

For some applications it is practical for a programmer to define his own input and output drivers, i.e. routines which perform input and output of characters to and from external devices. The following drivers are part of the TURBO environment, and used by the standard I/O drivers (although they are not available as standard procedures or functions):

**function** *ConSt*: boolean;
**function** *ConIn*: Char;
**procedure** *ConOut* (Ch: Char);
**procedure** *LstOut* (Ch: Char);
**procedure** *AuxOut* (Ch: Char);
**function** *AuxIn*: Char;
**procedure** *UsrOut* (Ch: Char);
**function** *UsrIn*: Char;





The *ConSt* routine is called by the function *KeyPressed*, the *ConIn* and *ConOut* routines are used by the CON:, TRM:, and KBD: devices, the *LstOut* routine is used by the LST: device, the *AuxOut* and *AuxIn* routines are used by the AUX: device, and the *UsrOut* and *UsrIn* routines are used by the USR: device.

By default, these drivers use the corresponding BIOS entry points of the CP/M operating system, i.e. *ConSt* uses CONST, *ConIn* uses CONIN, *ConOut* uses CONOUT, *LstOut* uses LIST, *AuxOut* uses PUNCH, *AuxIn* uses READER, *UsrOut* uses CONOUT, and *UsrIn* uses CONIN. This, however, may be changed by the programmer by assigning the address of a self-defined driver procedure or a driver function to one of the following standard variables:

| Variable | Contains the address of the |
|---|---|
| *ConStPtr* | *ConSt* function |
| *ConInPtr* | *ConIn* function |
| *ConOutPtr* | *ConOut* procedure |
| *LstOutPtr* | *LstOut* procedure |
| *AuxOutPtr* | *AuxOut* procedure |
| *AuxInPtr* | *AuxIn* function |
| *UsrOutPtr* | *UsrOut* procedure |
| *UsrInPtr* | *UsrIn* function |

A user defined driver procedure or driver function must match the definitions given above, i.e. a *ConSt* driver must be a *Boolean* function, a *ConIn* driver must be a *Char* function, etc.



ClibPDF - www.fastio.com

## External Subprograms

The reserved word **external** is used to declare external procedures and functions, typically procedures and functions written in machine code.

An external subprogram has no *block*, i.e. no declaration part and no statement part. Only the subprogram heading is specified, immediately followed by the reserved word **external** and an integer constant defining the memory address of the subprogram:

```
procedure DiskReset; external $EC00;
function IOstatus: boolean; external $D123
```

Parameters may be passed to external subprograms, and the syntax is exactly the same as that of calls to ordinary procedures and functions:

```
procedure Plot(X,Y: Integer); external $F003;
procedure QuickSort(var List: PartNo); external $1C00;
```

Parameter passing to external subprograms is discussed further on page 283.

## In-line Machine Code

TURBO Pascal features the **inline** statements as a very convenient way of inserting machine code instructions directly into the program text. An inline statement consists of the reserved word **inline** followed by one or more *code elements* separated by slashes and enclosed in parentheses.

A code element is built from one or more data elements, separated by plus (+) or minus (−) signs. A data element is either an integer constant, a variable identifier, a procedure identifier, a function identifier, or a location counter reference. A location counter reference is written as an asterisk (*).

**Example:**

```
inline (10/$2345/count+1/sort-*+2);
```





Each code element generates one byte or one word (two bytes) of code. The value of the byte or the word is calculated by adding or subtracting the values of the data elements according to the signs that separate them. The value of a variable identifier is the address (or offset) of the variable. The value of a procedure or function identifier is the address (or offset) of the procedure or function. The value of a location counter reference is the address (or offset) of the location counter, i.e. the address at which to generate the next byte of code.

A code element will generate one byte of code if it consists of integer constants only, and if its value is within the 8-bit range (0..255). If the value is outside the 8-bit range, or if the code element refers to variable, procedure, or function identifiers, or if the code element contains a location counter reference, one word of code is generated (least significant byte first).

The '<' and '>' characters may be used to override the automatic size selection described above. If a code element starts with a '<' character, only the least significant byte of the value is coded, even if it is a 16-bit value. If a code element starts with a '>' character, a word is always coded, even though the most significant byte is zero.

**Example:**

```
inline (<$1234/>$44);
```

This **inline** statement generates three bytes of code: $34, $44, $00.

The following example of an inline statement generates machine code that will convert all characters in its string argument to upper case.



ClibPDF - www.fastio.com

```
procedure UpperCase(var Strg: Str); {Str is type String[255]}
{$A+}
begin
  inline ($2A/Strg/           {       LD    HL,(Strg)   }
          $46/                {       LD    B,(HL)      }
          $04/                {       INC   B           }
          $05/                { L1:   DEC   B           }
          $CA/*+20/           {       JP    Z,L2        }
          $23/                {       INC   HL          }
          $7E/                {       LD    A,(HL)      }
          $FE/$61/            {       CP    'a'         }
          $DA/*-9/            {       JP    C,L1        }
          $FE/$7B/            {       CP    'z'+1       }
          $D2/*-14/           {       JP    NC,L1       }
          $D6/$20/            {       SUB   20H         }
          $77/                {       LD    (HL),A      }
          $C3/*-20);          {       JP    L1          }
                              { L2:   EQU   $           }
end;
```

**Inline** statements may be freely mixed with other statements throughout the statement part of a block, and **inline** statements may use all CPU registers. **Note**, however, that the contents of the stack pointer register (SP) must be the same on exit as on entry.





# Interrupt Handling

The TURBO Pascal run time package and the code generated by the compiler are both fully interruptable. Interrupt service routines must preserve all registers used.

If required, interrupt service procedures may be written in Pascal. Such procedures should always be compiled with the **A** compiler directive active ({$A+}), they must not have parameters, and they must themselves insure that all registers used are preserved. This is done by placing an **inline** statement with the necessary PUSH instructions at the very beginning of the procedure, and another **inline** statement with the corresponding POP instructions at the very end of the procedure. The last instruction of the ending **inline** statement should be an EI instruction ($FB) to enable further interrupts. If daisy chained interrupts are used, the **inline** statement may also specify a RETI instruction ($ED, $4D), which will override the RET instruction generated by the compiler.

The general rules for register usage are that integer operations use only the AF, BC, DE, and HL registers, other operations may use IX and IY, and real operations use the alternate registers.

An interrupt service procedure should not employ any I/O operations using the standard procedures and functions of TURBO Pascal, as these routines are not re-entrant. Also note that BDOS calls (and in some instances BIOS calls, depending on the specific CP/M implementation) should not be performed from interrupt handlers, as these routines are not re-entrant.

The programmer may disable and enable interrupts throughout a program using DI and EI instructions generated by **inline** statements.

If mode 0 (IM 0) or mode 1 (IM 1) interrupts are employed, it is the responsibility of the programmer to initialize the restart locations in the base page (note that RST 0 cannot be used, as CP/M uses locations 0 through 7).

If mode 2 (IM 2) interrupts are employed, the programmer should generate an initialized jump table (an array of integers) at an absolute address, and initialize the I register through a **inline** statement at the beginning of the program.



ClibPDF - www.fastio.com

# Internal Data Formats

In the following descriptions, the symbol @ denotes the address of the first byte occupied by a variable of the given type. The standard function *Addr* may be used to obtain this value for any variable.

## *Basic Data Types*

The basic data types may be grouped into structures (arrays, records, and disk files), but this structuring will not affect their internal formats.

### *Scalars*

The following scalars are all stored in a single byte: *Integer* subranges with both bounds in the range 0..255, *Booleans, Chars*, and declared scalars with less than 256 possible values. This byte contains the ordinal value of the variable.

The following scalars are all stored in two bytes: *Integers*, *Integer* subranges with one or both bounds not within the range 0..255, and declared scalars with more than 256 possible values. These bytes contain a 2's complement 16-bit value with the least significant byte stored first.

### *Reals*

Reals occupy 6 bytes, giving a floating point value with a 40-bit mantissa and an 8-bit 2's exponent. The exponent is stored in the first byte and the mantissa in the next five bytes which the least significant byte first:

| | |
|---|---|
| @ | Exponent |
| @ +1 | LSB of mantissa |
| : | |
| @ +5 | MSB of mantissa |

The exponent uses binary format with an offset of $80. Hence, an exponent of $84 indicates that the value of the mantissa is to be multiplied by 2 ^($84-$80) = 2 ^4 = 16. If the exponent is zero, the floating point value is considered to be zero.





The value of the mantissa is obtained by dividing the 40-bit unsigned integer by 2 ^40. The mantissa is always normalized, i.e. the most significant bit (bit 7 of the fifth byte) should be interpreted as a 1. The sign of the mantissa is stored in this bit, a 1 indicating that the number is negative, and a 0 indicating that the number is positive.

### *Strings*

A string occupies the number of bytes corresponding to one plus the maximum length of the string. The first byte contains the current length of the string. The following bytes contain the actual characters, with the first character stored at the lowest address. In the table shown below, **L** denotes the current length of the string, and *Max* denotes the maximum length:

| | |
|---|---|
| @ | Current length (*L*) |
| @ +1 | First character |
| @ +2 | Second character |
| : | |
| @ +*L* | Last character |
| @ +*L*+1 | Unused |
| : | |
| @ + *Max* | Unused |

### *Sets*

An element in a **set** occupies one bit, and as the maximum number of elements in a set is 256, a set variable will never occupy more than 32 bytes (256/8).

If a set contains less than 256 elements, some of the bits are bound to be zero at all times and need therefore not be stored. In terms of memory efficiency, the best way to store a set variable of a given type would then be to "cut off" all insignificant bits, and rotate the remaining bits so that the first element of the set would occupy the first bit of the first byte. Such rotate operations, however, are quite slow, and TURBO therefore employs a compromise: Only bytes which are statically zero (i.e. bytes of which no bits are used) are not stored. This method of compression is very fast and in most cases as memory efficient as the rotation method.



ClibPDF - www.fastio.com

The number of bytes occupied by a set variable is calculated as (*Max* **div** 8) − (*Min* **div** 8) + 1, where *Max* and *Min* are the upper and lower bounds of the base type of that set. The memory address of a specific element *E* is:

*MemAddress* = @ + (*E* **div** 8) − (*Min* **div** 8)

and the bit address within the byte at MemAddress is:

*BitAddress* = *E* **mod** 8

where *E* denotes the ordinal value of the element.

## File Interface Blocks

The table below shows the format of a FIB in TURBO Pascal-80:

| | |
|---|---|
| @ + 0 | Flags byte. |
| @ + 1 | Character buffer. |
| @ + 2 | Sector buffer pointer (LSB). |
| @ + 3 | Sector buffer pointer (MSB). |
| @ + 4 | Number of records (LSB). |
| @ + 5 | Number of records (MSB). |
| @ + 6 | Record length (LSB). |
| @ + 7 | Record length (MSB). |
| @ + 8 | Current record (LSB). |
| @ + 9 | Current record (MSB). |
| @ + 10 | Unused. |
| @ + 11 | Unused. |
| @ + 12 | First byte of CP/M FCB. |
| : | |
| @ + 47 | Last byte of CP/M FCB. |
| @ + 48 | First byte of sector buffer. |
| : | |
| @ + 175 | Last byte of sector buffer. |

The format of the flags byte at @ + 0 is:

| | |
|---|---|
| Bit 0..3 | File type. |
| Bit 4 | Read semaphore. |
| Bit 5 | Write semaphore. |
| Bit 6 | Output flag. |
| Bit 7 | Input flag. |





File type 0 denotes a disk file, and 1 through 5 denote the TURBO Pascal logical I/O devices (CON:, KBD:, LST:, AUX:, and USR:). For typed files, bit 4 is set if the contents of the sector buffer is undefined, and bit 5 is set if data has been written to the sector buffer. For textfiles, bit 5 is set if the character buffer contains a pre-read character. Bit 6 is set if output is allowed, and bit 7 is set if input is allowed.

The sector buffer pointer stores an offset (0..127) in the sector buffer at @ + 48. For typed and untyped files, the three words from @ + 4 to @ + 9 store the number of records in the file, the record length in bytes, and the current record number. The FIB of an untyped file has no sector buffer, and so the sector buffer pointer is not used.

When a text file is assigned to a logical device, only the flags byte and the character buffer are used.

## Pointers

A pointer consists of two bytes containing a 16-bit memory address, and it is stored in memory using byte reversed format, i.e. the least significant byte is stored first. The value **nil** corresponds to a zero word.

## Data Structures

Data structures are built from the basic data types using various structuring methods. Three different structuring methods exist: arrays, records, and disk files. The structuring of data does not in any way affect the internal formats of the basic data types.

## Arrays

The components with the lowest index values are stored at the lowest memory address. A multi-dimensional array is stored with the rightmost dimension increasing first, e.g. given the array

```
Board: array[1..8,1..8] of Square
```

you have the following memory layout of its components:



ClibPDF - www.fastio.com

```
lowest address:  Board[1,1]
                 Board[1,2]
                 :
                 Board[1,8]
                 Board[2,1]
                 Board[2,2]
                 :
                 :
Highest address: Board[8,8]
```

## Records

The first field of a record is stored at the lowest memory address. If the record contains no variant parts, the length is given by the sum of the lengths of the individual fields. If a record contains a variant, the total number of bytes occupied by the record is given by the length of the fixed part plus the length of largest of its variant parts. Each variant starts at the same memory address.

## Disk Files

Disk files are different from other data structures in that data is not stored in internal memory but in a file on an external device. A disk file is controlled through a file interface block (FIB) as described on page 280. In general there are two different types of disk files: random access files and text files.

### Random Access Files

A random access file consists of a sequence of records, all of the same length and same internal format. To optimize file storage capacity, the records of a file are totally contiguous. The first four bytes of the first sector of a file contains the number of records in the file and the length of each record in bytes. The first record of the file is stored starting at the fourth byte.

| | |
|---|---|
| sector 0, byte 0: | Number of records (LSB) |
| sector 0, byte 1: | Number of records (MSB) |
| sector 0, byte 2: | Record length (LSB) |
| sector 0, byte 3: | Record length (MSB) |





### Text Files

The basic components of a text file are characters, but a text file is subdivided into *lines*. Each line consists of any number of characters ended by a CR/LF sequence (ASCII $0D/ $0A). The file is terminated by a Ctrl-Z (ASCII $1A).

## Parameters

Parameters are transferred to procedures and functions via the Z-80 stack. Normally, this is of no interest to the programmer, as the machine code generated by TURBO Pascal will automatically PUSH parameters onto the stack before a call, and POP them at the beginning of the subprogram. However, if the programmer wishes to use **external** subprograms, these must POP the parameters from the stack themselves.

On entry to an **external** subroutine, the top of the stack always contains the return address (a word). The parameters, if any, are located below the return address, i.e. at higher addresses on the stack. Therefore, to access the parameters, the subroutine must first POP off the return address, then all the parameters, and finally it must restore the return address by PUSHing it back onto the stack.

## Variable Parameters

With a variable (VAR) parameter, a word is transferred on the stack giving the absolute memory address of the first byte occupied by the actual parameter.

## Value Parameters

With value parameters, the data transferred on the stack depends upon the type of the parameter as described in the following sections.

### Scalars

*Integers, Booleans, Chars* and declared scalars are transferred on the stack as a word. If the variable occupies only one byte when it is stored, the most significant byte of the parameter is zero. Normally, a word is POPped off the stack using an instruction like POP HL.



ClibPDF - www.fastio.com

### Reals

A real is transferred on the stack using six bytes. If these bytes are POPped using the instruction sequence:

```
POP     HL
POP     DE
POP     BC
```

then L will contain the exponent, H the fifth (least significant) byte of the mantissa, E the fourth byte, D the third byte, C the second byte, and B the first (most significant) byte.

### Strings

When a string is at the top of the stack, the byte pointed to by SP contains the length of the string. The bytes at addresses SP + 1 through SP + n (where n is the length of the string) contain the string with the first character stored at the lowest address. The following machine code instructions may be used to POP the string at the top of the stack and store it in *StrBuf*:

```
LD      DE,StrBuf
LD      HL,0
LD      B,H
ADD     HL,SP
LD      C,(HL)
INC     BC
LDIR
LD      SP,HL
```

### Sets

A set always occupies 32 bytes on the stack (set compression only applies to the loading and storing of sets). The following machine code instructions may be used to POP the set at the top of the stack and store it in *SetBuf*:





```
LD      DE,SetBuf
LD      HL,0
ADD     HL,SP
LD      BC,32
LDIR
LD      SP,HL
```

This will store the least significant byte of the set at the lowest address in *SetBuf*.

### Pointers

A pointer value is transferred on the stack as a word containing the memory address of a dynamic variable. The value NIL corresponds to a zero word.

### Arrays and Records

Even when used as value parameters, *Array* and *Record* parameters are not actually PUSHed onto the stack. Instead, a word containing the address of the first byte of the parameter is transferred. It is then the responsibility of the subroutine to POP this word, and use it as the source address in a block copy operation.

## Function Results

User written **external** functions must return their results exactly as specified in the following:

Values of scalar types, must be returned in the HL register pair. If the type of the result is expressed in one byte, then it must be returned in L and H must by zero.

*Reals* must be returned in the BC, DE, and HL register pairs. B, C, D, E, and H must contain the mantissa (most significant byte in B), and L must contain the exponent.

**Strings** and **sets** must be returned on the top of the stack on the formats described on page 284.

Pointer values must be returned in the HL register pair.



ClibPDF - www.fastio.com

## The Heap and The Stacks

As indicated by the memory maps in previous sections, three stack-like structures are maintained during execution of a program: The *heap*, the *CPU stack*, and the *recursion stack*.

The heap is used to store dynamic variables, and is controlled with the standard procedures *New, Mark,* and *Release*. At the beginning of a program, the heap pointer *HeapPtr* is set to the address of the bottom of free memory, i.e the first free byte after the object code.

The CPU stack is used to store intermediate results during evaluation of expressions and to transfer parameters to procedures and functions. An active **for** statement also uses the CPU stack, and occupies one word. At the beginning of a program, the CPU stack pointer *StackPtr* is set to the address of the top of free memory.

The recursion stack is used only by recursive procedures and functions, i.e. procedures and functions compiled with the **A** compiler directive passive ({$A-}). On entry to a recursive subprogram it copies its workspace onto the recursion stack, and on exit the entire workspace is restored to its original state. The default initial value of *RecurPtr* at the beginning of a program, is 1K ($400) bytes below the CPU stack pointer.

Because of this technique, variables local to a subprogram must not be used as **var** parameters in recursive calls.

The pre-defined variables:

| | |
|---|---|
| *HeapPtr:* | The heap pointer, |
| *RecurPtr:* | The recursion stack pointer, and |
| *StackPtr:* | The CPU stack pointer |

allow the programmer to control the position of the heap and the stacks.

The type of these variables is *Integer*. Notice that *HeapPtr* and *RecurPtr* may be used in the same context as any other *Integer* variable, whereas *StackPtr* may only be used in assignments and expressions.

When these variables are manipulated, always make sure that they point to addresses within free memory, and that:

*HeapPtr* < *RecurPtr* < *StackPtr*





Failure to adhere to these rules will cause unpredictable, perhaps fatal, results.

Needless to say, assignments to the heap and stack pointers must never occur once the stacks or the heap are in use.

On each call to the procedure *New* and on entering a recursive procedure or function, the system checks for collision between the heap and the recursion stack, i.e. checks if *HeapPtr* is less than *RecurPtr*. If not, a collision has occurred, which results in an execution error.

Note that **no** checks are made at any time to insure that the CPU stack does not overflow into the bottom of the recursion stack. For this to happen, a recursive subroutine must call itself some 300-400 times, which must be considered a rare situation. If, however, a program requires such nesting, the following statement executed at the beginning of the program block will move the recursion stack pointer downwards to create a larger CPU stack:

```
RecurPtr := StackPtr -2 *MaxDepth -512;
```

where *MaxDepth* is the maximum required depth of calls to the recursive subprogram(s). The extra approx. 512 bytes are needed as a margin to make room for parameter transfers and intermediate results during the evaluation of expressions.



ClibPDF - www.fastio.com

# Memory Management

## Memory Maps

The following diagrams illustrate the contents of memory at different stages of working with the TURBO system. Solid lines indicate fixed boundaries (i.e. determined by amount of memory, size of your CP/M, version of TURBO, etc.), whereas dotted lines indicate boundaries which are determined at run-time (e.g. by the size of the source text, and by possible user manipulation of various pointers, etc.). The sizes of the segments in the diagrams do not necessarily reflect the amounts of memory actually consumed.

### Compilation in Memory

During compilation of a program in memory (**M**emory-mode on compiler **O**ptions menu, see page 259), the memory is mapped as follows:

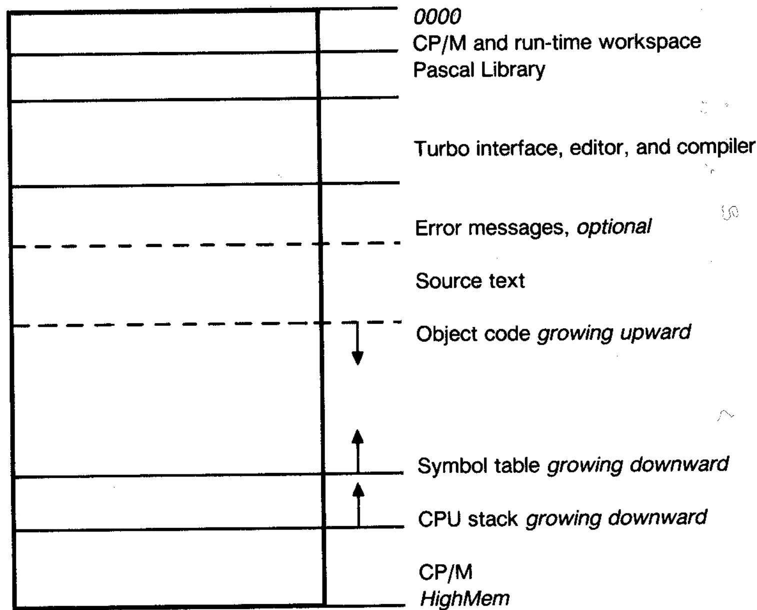


*Figure 22-5: Memory map during compilation in memory*





If the error message file is not loaded when starting TURBO, the source text starts that much lower in memory. When the compiler is invoked, it generates object code working upwards from the end of the source text. The CPU stack works downwards from the logical top of memory, and the compiler's symbol table works downwards from an address 1K ($400 bytes) below the logical top of memory.

### Compilation To Disk

During compilation to a .COM or .CHN file (**C**om-mode or c**H**n-mode on compiler **O**ptions menu, see page 259), the memory looks much as during compilation in memory (see preceding section) *except* that generated object code does not reside in memory but is written to a disk file. Also, the code starts at a higher address (right after the Pascal library instead of after the source text). Compilation of much larger programs is thus possible in this mode.

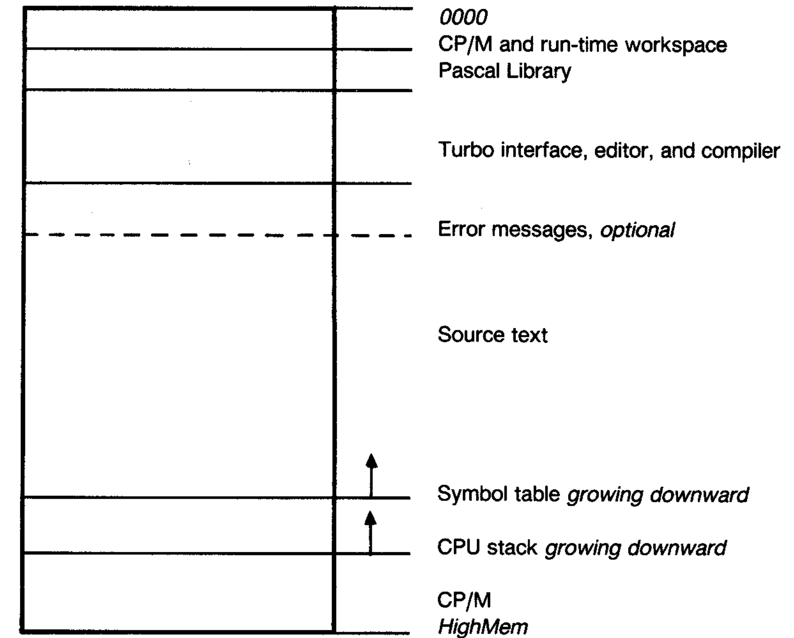


*Figure 22-6: Memory map during compilation to a file*



ClibPDF - www.fastio.com

## Execution in Memory

When a program is executed in direct - or memory - mode (i.e. the **M**emory-mode on compiler **O**ptions menu is selected, see page 259), the memory is mapped as follows:

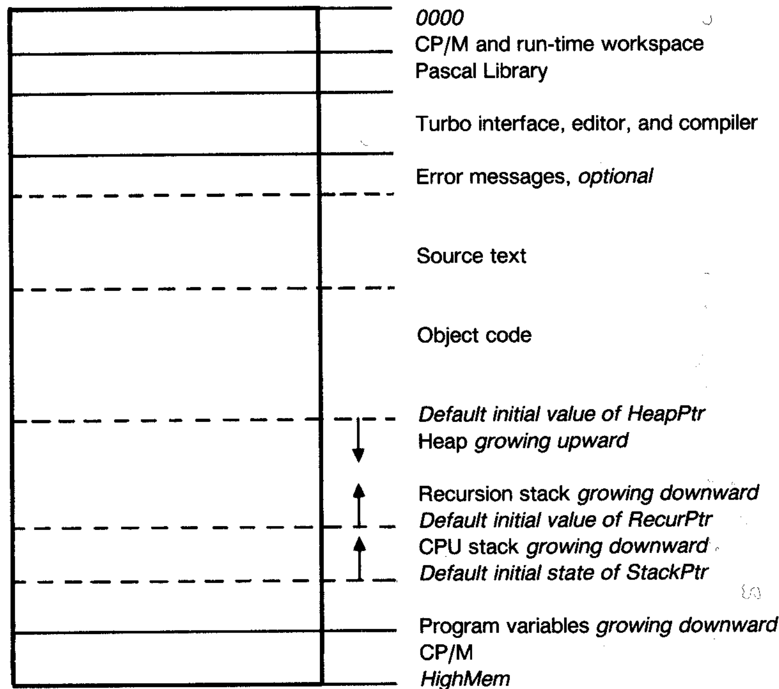


*Figure 22-7: Memory map during execution in direct mode*

When a program is compiled, the end of the object code is known. The heap pointer *HeapPtr* is set to this value by default, and the heap grows from here and upwards in memory towards the recursion stack. The maximum memory size is BDOS minus one (indicated on the compiler **O**ptions menu). Program variables are stored from this address and downwards. The end of the variables is the 'top of free memory' which is the initial value of the CPU stack pointer *StackPtr*. The CPU stack grows downwards from here towards the position of the recursion stack pointer *RecurPtr*, $400 bytes lower than *StackPtr*. The recursion stack grows from here downward towards the heap.





## Execution of A Program File

When a program file is executed (either by the **R**un command with the **M**emory-mode on the compiler **O**ptions menu selected, by an e**X**ecute command, or directly from CP/M), the memory is mapped as follows:

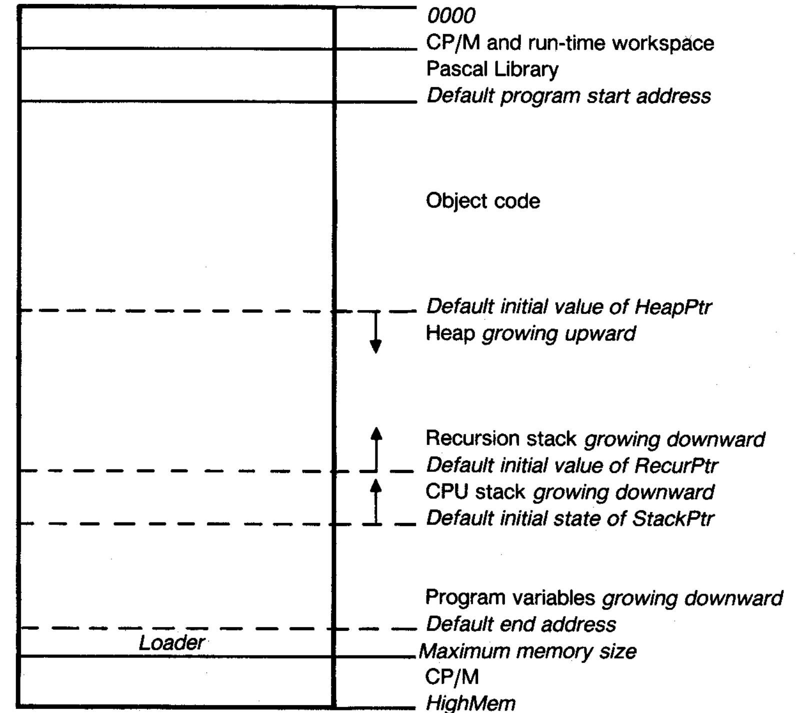


*Figure 22-8: Memory map during execution of a program file*

This map resembles the previous, except for the absence of the TURBO interface, editor, and compiler (and possible error messages) and of the source text. The *default program start address* (shown on the compiler **O**ptions menu) is the first free byte after the Pascal runtime library. This value may be manipulated with the **S**tart address command of the compiler **O**ptions menu, e.g. to create space for **absolute** variables and/or external procedures between the library and the code. The *maximum memory size* is BDOS minus one, and the default value is determined by the BDOS location on the computer in use.



ClibPDF - www.fastio.com

If programs are to be translated for other systems, care should be taken to avoid collision with the BDOS. The maximum memory may be manipulated with the **E**nd address command of the compiler **O**ptions menu. Notice that the *default end address* setting is approx. 700 to 1000 bytes lower than maximum memory. This is to allow space for the loader which resides just below BDOS when .COM files are **R**un or e**X**ecuted from the TURBO system. This loader restores the TURBO editor, compiler, and possible error messages when the program finishes and thus returns control to the TURBO system.





# Chapter 23
# TURBO-BCD

TURBO-BCD is a special version of TURBO Pascal which is not included in the standard TURBO Pascal package. It employs binary coded decimal (BCD) *Real* numbers to obtain higher accuracy, especially needed in programs for business applications.

If you are interested in purchasing TURBO-BCD, please see page 3 for ordering information.

TURBO-BCD will compile and run any program written for standard TURBO or TURBO-87 Pascal; the only difference being in real number processing and real number format.

## Files On the TURBO-BCD Distribution Diskette

In addition to the files listed on page 8, the TURBO-BCD distribution diskette contains the file

**TURBOBCD.COM**

(TURBOBCD.CMD for CP/M-86). This file contains the special TURBO-BCD system. If you want to install it with TINST, you must first temporarily rename it to TURBO.COM (or .CMD).

## BCD Range

TURBO-BCD's BCD *Reals* have a range of 1E-63 through 1E + 63 with 18 significant digits.



ClibPDF - www.fastio.com

## Form function

Syntax: Form(*St, Var1, Var2,.., VarN*)

The *Form* function provides advanced numeric and string formatting. *St* is a string expression giving an image of the format string, as detailed in the following, and *Var1,Var2,..,VarN* are *Real*, *Integer*, or *String* expressions. The result is a *String* of the same length as *St*.

*St* is made up of a number of field specifiers, each of which corresponds to one parameter in the parameter list. Blanks and characters other than the ones defined in the following serve to separate fields and will also appear in the formatted result, viz:

```
Form('Total: $#,###.##',1234.56) = 'Total: $1,234.56'
```

The arguments in the argument list use the field specifiers in the order of appearance:

```
Form('Please @@@@@@ us at (###) ### ####','phone',408,438,8400) =
'Please  phone us at (408) 438 8400 '
```

If there are more arguments in the argument list than there are field specifiers in the format string, the arguments in excess are ignored. If there are less arguments than field specifiers, the field specifiers in excess are returned unchanged:

```
Form('###.##',12.34,43.21) = ' 12.34'
Form('###.## -##.##',123.4) = '123.40 -##.##'
```

There are two types of field specifiers: **numeric** and **string**.

### *Numeric Fields*

A numeric field is a sequence of one or more of the following characters:

```
#  @  *  $  -  +  ,  .
```





Any other character terminates the numeric field. The number is returned right-justified within the field, decimals are rounded if they exceed the number of decimals specified by the format, and if the number is too large to be returned in the field, all digit positions are filled with asterisks.

\# A digit position. If the numeric field contains no @ or * characters, unused digits are returned as blanks. If the numeric field contains no sign positions ('-' or '+' characters) and the number is negative, a floating minus is returned in front of the number.

**Examples:**

```
Form('####',34.567)     = '  35'
Form('###.##',12.345')  = ' 12.35'
Form('####.##',-12.3)   = ' -12.30'
Form('###.##',1234.5)   = '***.**'
```

@ A digit position. Unused digits are forced to be returned as zeros instead of blanks. The @ character needs only occur once in the numeric field to activate this effect. The sign of the number will not be returned unless the field contains a sign position ('-' or '+' character).

**Examples:**

```
Form('@##',9)          = '009'
Form('@@@.@@',12.345)  = '012.35'
```

* A digit position. Unused digits are forced to be returned as asterisks instead of blanks. The * character needs only occur once in the numeric field to activate this effect. The sign of the number will not be returned unless the field contains a sign position ('-' or '+' character).

**Examples:**

```
Form('*##.#',4.567)  = '**4.57'
Form('****',123)     = '*123'
```



ClibPDF - www.fastio.com

$ A digit position. A floating $-sign is returned in front of the number. The '$' character need only occur once in the numeric field to activate this effect.

**Examples:**
```
Form('$#####.##',123.45)  = '  $123.45'
Form('######.#$',-12.345) = '  -$12.35'
Form('*$####.##',12.34)   = '***$12.34'
```

- A sign position. If the number is negative, a minus will be returned in that position; if it is positive, a blank is returned.

**Examples:**
```
Form('-###.##',-1.2)         = '-  1.20'
Form('-###.##',12)           = '  12.00'
Form('*#####.##-',-123.45) = '***123.45-'
```

\+ A sign position. If the number is positive, a plus will be returned in that position; if it is negative, a minus is returned.

**Examples:**
```
Form('+###.##',-1.2)       = '-  1.20'
Form('+###.##',12)         = '+ 12.00'
Form('*$####.##+',12.34) = '***$12.34+'
```

, A decimal comma or a separator comma. The last period or comma in the numeric image is considered the decimal delimiter.

. A decimal period or a separator period. The last period or comma in the numeric image is considered the decimal delimiter.

**Examples:**
```
Form('##,###,###.##',12345.6)    = '    12,345.60'
Form('$#.###.###,##',-12345.6)  = '  -$12.345,60'
Form('*$,###,###.##+',12345.6)  = '***$12,345.60+'
Form('##,###.##',123456.0)       = '**,***.**'
```





## String Fields

A string field is a sequence of # or @ characters. If the string parameter is longer than the string field, only the first characters of the string are returned.

\# If the field contains only # characters, the string will be returned left justified.

@ If one or more '@' characters are present in the field, the string will be returned right justified within the length of the field.

**Examples:**

```
Form('##########','Pascal') = 'Pascal    '
Form('@#########','Pascal') = '    Pascal'
Form('####','TURBO Pascal') = 'TURBO '
Form('@@@@','TURBO Pascal') = 'TURBO '
```

## Writing BCD Reals

BCD *Reals* are written on a format slightly different from the standard format, as described below.

*R* The decimal representation of the value of *R* is output in a field 25 characters wide, using floating point format. For *R* > = 0.0, the format is:

```
␣␣#.################E*##
```

For *R* < 0.0, the format is:

```
␣-#.################E*##
```

where ␣ represents a blank, # represents a digit, and * represents either plus or minus.



ClibPDF - www.fastio.com

*R:n* The decimal representation of the value of *R* is output, right adjusted in a field *n* characters wide, using floating point format. For *R* > = 0.0:

```
blanks#.digitsE*##
```

For *R* < 0.0:

```
blanks-#.digitsE*##
```

where *blanks* represents zero or more blanks, *digits* represents from 1 to 17 digits, # represents a digit, and * represents either plus or minus.

## Formatted Writing

The *Form* standard function can be used as a *write parameter* to produce formatted output:

```
Write(Form('The price is $###,###,###.##',Price));
```

## Internal Data Format

The BCD *Real* variable occupies 10 bytes, and consists of a floating point value with an 18 digit binary coded decimal mantissa, a 7-bit 10's exponent, and a 1-bit sign. The exponent and the sign are stored in the first byte and the mantissa in the next nine bytes with the least significant byte first:

```
@+0      Exponent and sign.
@+1      LSB of mantissa.
 :
@+9      MSB of mantissa.
```





The most significant bit of the first byte contains the sign. 0 means positive and 1 means negative. The remaining seven bits contain the exponent in binary format with an offset of $3F. Thus, an exponent of $41 indicates that the value of the mantissa is to be multiplied by 10^($41-$3F) = 10^2 = 100. If the first byte is zero, the floating point value is considered to be zero. Starting with the tenth byte, each byte of the mantissa contains two digits in BCD format, with the most significant digit in the upper four bits. The first digit contains the 1/10's, the second contains the 1/100's, etc. The mantissa is always normalized, i.e. the first digit is never 0 unless the entire number is 0.

This 10-byte *Real* is not compatible with TURBO standard or 8087 *Reals*. This, however, should only be a problem if you develop programs in different versions of TURBO which must interchange data. The trick then is simply to provide an interchange-format between the programs in which you transfer *Reals* on ASCII format, for instance.



ClibPDF - www.fastio.com

**Notes:**





# Chapter 24
# TURBO-87

*TURBO-87*

TURBO-87 is a special version of TURBO Pascal which is not included in the standard TURBO Pascal package. It uses the Intel 8087 math-processor for real number arithmetic, providing a significant gain in speed. TURBO-87 does not include the 8087 chip.

If you are interested in purchasing TURBO-87, please see page 3 for ordering information.

TURBO-87 will compile and run any program written for standard TURBO Pascal; the only difference being in real number processing and real number format.

TURBO-87 programs will **not** run on a computer without the 8087-chip installed, whereas the opposite will work.

## Files On the TURBO-87 Distribution Diskette

In addition to the files listed on page 8, the TURBO-87 distribution diskette contains the file

**TURBO-87.COM**

(TURBO-87.CMD for CP/M-86). This file contains the special TURBO-87 system. If you want to install it with TINST, you must first temporarily rename it to TURBO.COM (or .CMD).



ClibPDF - www.fastio.com

## Writing 8087 Reals

8087 *Reals* are written on a format slightly different from the standard format, as described below.

*R* The decimal representation of the value of *R* is output in a field 23 characters wide, using floating point format. For *R* > = 0.0, the format is:

```
␣␣#.################E*##
```

For *R* < 0.0, the format is:

```
␣-#.################E*##
```

where ␣ represents a blank, # represents a digit, and * represents either plus or minus.

*R:n* The decimal representation of the value of *R* is output, right adjusted in a field *n* characters wide, using floating point format. For *R* > = 0.0:

```
blanks#.digitsE*##
```

For *R* < 0.0:

```
blanks-#.digitsE*##
```

where *blanks* represents zero or more blanks, *digits* represents from 1 to 14 digits, # represents a digit, and * represents either plus or minus.

## Internal Data Format

The 8087 chip supports a range of data types. The one used by TURBO-87 is the *long real*; its 64-bits yielding 16 digits accuracy and a range of 4.19E-307 to 1.67E + 308.

This 8-byte *Real* is not compatible with TURBO standard or BCD *Reals*. This, however, should only be a problem if you develop programs in different versions of TURBO which must interchange data. The trick then is simply to provide an interchange-format between the programs in which you transfer *Reals* on ASCII format, for instance.





# Appendix A
# SUMMARY OF STANDARD PROCEDURES AND FUNCTIONS

This appendix lists all standard procedures and functions available in TURBO Pascal and describes their application, syntax, parameters, and type. The following symbols are used to denote elements of various types:

| | |
|---|---|
| *type* | any type |
| *string* | any string type |
| *file* | any file type |
| *scalar* | any scalar type |
| *pointer* | any pointer type |

Where parameter type specification is not present, it means that the procedure or function accepts variable parameters of any type.

## Input/Output Procedures and Functions

The following procedures use a non-standard syntax in their parameter lists:

**procedure**
Read (**var** *F*: **file of** *type*; **var** V: *type*);
Read (**var** *F*: text; **var** *I*: Integer);
Read (**var** *F*: text; **var** *R*: Real);
Read (**var** *F*: text; **var** *C*: Char);
Read (**var** *F*: text; **var** *S*: *string*);
Readln (**var** *F*: text);
Write (**var** *F*: **file of** *type*; **var** V: *type*);
Write (**var** *F*: text; *I*: Integer);
Write (**var** *F*: text; *R*: Real);
Write (**var** *F*: text; *B*: Boolean);
Write (**var** *F*: text; *C*: Char);
Write (**var** *F*: text; *S*: *string*);
Writeln (**var** *F*: text);



ClibPDF - www.fastio.com