Turbo-51

Documentation

by Igor Funa

edited by
Jürgen Rathlev

Table of contents

1 Introduction

Turbo51 by Igor Funa is a free Pascal compiler for the 8051 family of microcontrollers. If you are programming for the 8051 family of microcontrollers and you like Pascal programming language then you will love Turbo51.

Main features:	**Used optimizations:**
• Win32 console application • Fast single-pass optimizing compiler • Borland Turbo Pascal 7 syntax • Full floating point support • Mixed Pascal and assembler code • Full use of register banks • Advanced multi-pass optimizer • Smart linker • Generates compact high quality code • Output formats: BIN, HEX, OMF • Assembler source code generation • Source-level debugging with absolute OMF-51 extended object file	• Constant folding • Integer arithmetic optimizations • Dead code elimination • Branch elimination • Code-block reordering • Loop-invariant code motion • Loop inversion • Induction variable elimination • Instruction selection • Instruction combining • Register allocation • Common subexpression elimination • Peephole optimization

Turbo51 is released as a freeware. You can use Turbo51 for hobby projects and for serious work. Check documentation pages and code examples that show the syntax, features and generated files. This should be enough to start a 8051 project development with Turbo51. And if you are still missing something or have a problem you can always ask for help.

If you are already familiar with 8051 assembly language programming you can start with Turbo51 as 8051 assembly language compiler and then add some Pascal statements until you become familiar with Turbo51 and Pascal syntax. A good approach is also to compile some Pascal code and then check generated code (ASM file). This way you can learn assembly language, get some ideas on how to write effective code and become familiar with the compiler. Turbo51, like many popular C compilers for 8051, generates optimized code and supports source-level debugging with OMF object file.

Turbo51 is a command line console application. This means that it has no graphical user interface, menus or windows. It must be run from a console with parameters: pascal source file to compile and optional switches. Optionally you can run Turbo51 from some Integrated Development Environment or editor which supports external applications. It is always a good idea to use the -M option to recompile modified units to u51 file. The -M switch recompiles only units that need to be recompiled, either their source was modified or one of their used units was modified in interface section or one of the used include files was changed. This is the fastest way of compilation. Optionally you can force to recompile all used units with the -B switch.

Turbo51 uses most of the Borland Turbo Pascal 7 syntax including OOP and some additional directives and constructs to support specific features of 8051 family (MCS-51).

Reserved words:

`AND, ARRAY, ASM, BEGIN, CASE, CONST, CONSTRUCTOR, DESTRUCTOR, DIV, DO, DOWNTO, ELSE, END, FILE, FOR, FUNCTION, GOTO, IF, IMPLEMENTATION, IN, INHERITED, INTERFACE, LABEL, MOD, NIL, NOT, OBJECT, OF, OR, PACKED, PROCEDURE, PROGRAM, RECORD, REPEAT, SET, SHL, SHR, STRING, THEN, TO, TYPE, UNIT, UNTIL, USES, VAR, WHILE, WITH, XOR`

Directives:

`ABSOLUTE, ASSEMBLER, BITADDRESSABLE, CODE, DATA, EXTERNAL, FORWARD, IDATA, INLINE, INTERRUPT, PRIVATE, PUBLIC, REENTRANT, USING, USINGANY, VIRTUAL, VOLATILE, XDATA`

This manual is derived from the internet site of Turbo-51: http://turbo51.com/documentation

2 General

2.1 Command line syntax

```
Turbo51 [options] filename [options]
```

Option	***Description***
-A	Generate assembler file
-B	Build
-C	Show error column number
-D*symbols*	Define conditionals
-E*path*	BIN/HEX/U51/OMF output directory
-F*hex address*	Find source line at address
-G	Generate map file
-H	Generate Intel HEX file
-I*path*	Include file directories
-J*path*	Object file directories
-LA	Use library Turbo51A.l51 (compiled with $A+, ACALL/AJMP instructions)
-M	Make modified units
-MG*memory type*	Set default memory type for global variables (memory type = D, I or X)
-ML*memory type*	Set default memory type for local variables (memory type = D, I or X)
-MP*memory type*	Set default memory type for parameter variables (memory type = D, I or X)
-MT*memory type*	Set default memory type for temporary variables (memory type = D, I or X)
-O	Generate OMF-51 file
-OX	Generate extended OMF-51 file (needed for source-level debugging)
-Q	Quiet compile
-S	Syntax check
-T*path*	L51/CFG directory
-U*path*	Unit file directories
-$*directive*	Command line compiler switch

It is a good idea to use command line option /M to compile used units only when they or files they depend on were changed. This can speed up the compilation. You can force Turbo51 to rebuild all used units with command line option /B. Each time the unit is compiled Turbo51 generates a file

'UnitName.u51'. This file is used next time when the main file is compiled. Without using command line options /M or /B Turbo51 will each time compile all used units without making the compiled unit files (u51).

Command line compiler switches:

(default values see 2.2)

Compiler switch	*Description*
-$A-	Generate absolute instructions (`ACALL/AJMP`)
-$B-	Full boolean evaluation
-$C+	Show source lines in assembler file
-$I+	IDATA variables can start below $80 (as indirectly addressed DATA variables)
-$O+	Optimizations
-$P-	Open string parameters
-$R-	Reentrant procedures
-$T-	Typed pointers
-$U-	Unique local variable names
-$V+	Strict var-strings
-$X+	Extended syntax

2.2 Switches and directives

Compiler switches: `{$<letter/switchname><state>[,<letter/switchname><state>] }`

(default values are shown below)

Compiler switch	*Description*
$A-	Generate absolute instructions (`ACALL/AJMP`)
$AbsoluteInstructions *Off*	dto.
$B-	Full boolean evaluation
$C+	Show source lines in assembler file
$DefaultFile *Off*	Assume *CurrentIO* system file variable is assigned with the actual IO procedures
$I+	IDATA variables can start below $80 (as indirectly addressed DATA variables)
$InlineCode *On*	If set to Off compiler generates normal call to inline procedure
$NoReturn	Inside assembler procedure prevents generation of `RET` instruction
$O+	Optimizations
$P-	Open string parameters
$R-	Reentrant procedures
$T-	Typed pointers
$U-	Unique local variable names
$V+	Strict var-strings
$X+	Extended syntax

Note: there is no space between switch letter and + or - and there is space between long switch name and On or Off.

Examples: `{$DefaultFile On },{$O+ },{$C+}`

Compiler directives: `{$<directive><value> }`

Compiler directive	*Description*
$DEFINE	Defines symbol
$ELSE	Conditional compilation with `IFDEF` and `IFNDEF`
$ENDIF	End conditional compilation
$IFDEF *symbol*	Conditional compilation if symbol is defined
$IFNDEF *symbol*	Conditional compilation if symbol is not defined
$IFOPT *switch*(+/-)	Conditional compilation if compiler switch is set/not set
$M	Memory sizes (only in program), default values:
CODE Start,	$0000
CODE Size,	$10000
XDATA Start,	$0000
XDATA Size,	$0000
Heap Size	$0000
$IDATA	IDATA memory available
$XDATA	XDATA memory available (only in unit)
$HEAP	Heap available (only in unit)
$MG *memory type*	Set default memory type for global variables (*memory type = DATA, IDATA* or *XDATA*)
$ML *memory type*	Set default memory type for local variables (*memory type = DATA, IDATA* or *XDATA*)
$MP *memory type*	Set default memory type for parameter variables (*memory type = DATA, IDATA* or *XDATA*)
$MT *memory type*	Set default mem. type for temporary variables (*memory type = DATA, IDATA* or *XDATA*)

Examples: `{$M $8000,$1000,$9000,$1000,$400}},{$XDATA}`

2.3 Memory organization

CODE memory

By default the start address of the code memory is $0000, the maximum code size is $10000 bytes (64 KB). This can be changed in the main program with the **`$M`** directive.

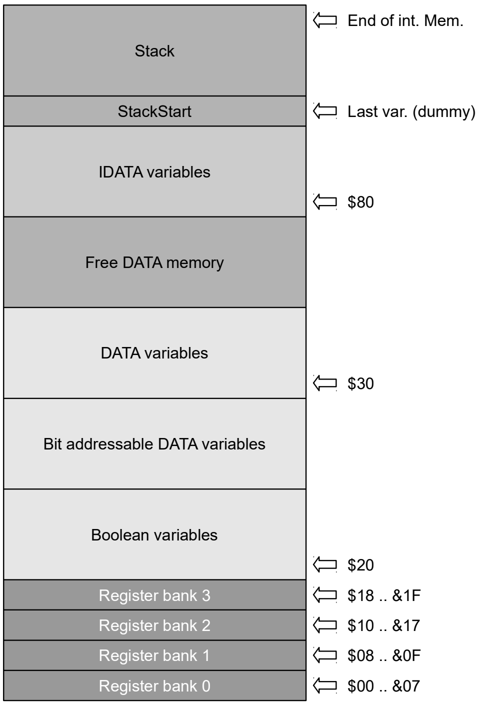

Fig. 1: Internal memory

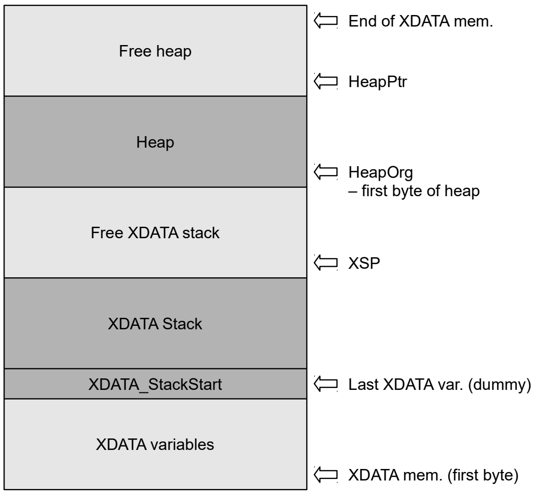

Fig. 2: External memory

IDATA memory

By default there is no IDATA memory. This can be changed either in the main program or in one of the directly or indirectly used units with the `$IDATA` directive. Usually the main program uses a unit which declares features of the microcontroller including the IDATA memory with the `$IDATA` directive.

XDATA memory

By default there is no XDATA memory. This can be changed either in the main program with the `$M` directive or in one of the used units with the `$XDATA` directive.

DATA / IDATA Memory organization

If compiler switch `$I` is set then IDATA variables immediately follow DATA variables. If there is no IDATA memory or IDATA variables then stack immediately follows DATA variables (Fig. 1).

XDATA Memory organization

Start address and size of XDATA memory and heap size can be set with compiler directive `{$MCODE Size, XDATA Start, XDATA Size, Heap Size}` (Fig. 2).

2.4 System unit

System unit implements Turbo51 runtime library and defines some special function registers (SFR), bits and interrupt addresses that are present in all microcontrollers based on 8051 core. It is implicitly used by the compiler. Usually it is loaded from the library (Turbo51.l51) which is a binary concatenation of units (currently only system unit is included).

There is also the Turbo51A.l51 library which contains the system unit compiled with the `$A+` switch and has no `LCALL/LJMP` instructions (to use it use the **-LA** command line option). Warning: until the compiler will reach some more stable state some declarations in this this unit might change.

Additional definitions for other microcontrollers of the 8051 family are found in several units with names like *Sys_xxxx.pas* (e.g. *Sys_80C592.pas*, *Sys_89S8253.pas*).

Find all definitions of the system unit in **Appendix A**.

2.5 Files

Turbo51 supports files - a general framework for IO handling. However, you have to provide the low level IO procedures. Files can be untyped, typed (File of SomeType) and of type Text (ASCII text terminated with line feed (#10) character). The following procedures support files:

- *Assign*
- *Read*
- *ReadLn*
- *BlockRead*
- *Write*
- *WriteLn*
- *BlockWrite*

See **Appendix B** for a simple example of a calculator using files.

2.6 Objects

Objects are data structures that merge pascal records and procedures called methods, i.e. data and code together. In order to use objects in Turbo51 you need XDATA memory. The syntax is equivalent to that in Borland Turbo Pascal 7. Turbo51 supports:

- Inheritance
- Static and dynamic objects
- Private fields
- Constructors and destructors
- Static, virtual and dynamic methods

See **Appendix C** for an example.

3 Declarations

3.1 Constants

Turbo51 constants can be of any ordinal type. Typed constants are stored in CODE memory (in little endian format) and can not be modified. Boolean typed constants are not possible because *boolean* data can only be stored as bits in bit-addressable DATA memory (which is available in all 8051 derivatives), but you can use *ByteBool* or similar typed constants.

Find some examples in **Appendix D**.

3.2 Types

Turbo51 provides the following system types: *Byte* (unsigned 8-bit), *Word* (unsigned 16-bit), *ShortInt* (signed 8-bit), *Integer* (signed 16-bit), *LongInt* (signed 32-bit), *Real* (uses 4 bytes), *String*, *Boolean*, *ByteBool*, *WordBool*, *LongBool* and *Char.* You can also construct any other type according to Pascal syntax. In Turbo51 there are three types of pointer: *ShortPtr* (points to IDATA), *Pointer* (points to XDATA) and *CodePointer* (points to CODE). Similarly there are *ShortPChar, PChar* and *CodePChar*. Pointers to ordinal types can have memory type directive DATA, IDATA or XDATA which overrides default memory type for variables and sets memory type to which this pointer will point to.

Find some examples in **Appendix E**.

3.3 Variables

Turbo51 variables can have memory type directives DATA, IDATA or XDATA which overrides default memory type for variables (IDATA memory with addresses starting from $80 is not available on all 8051 derivatives, some 8051 derivatives have also internal XDATA memory). Boolean variables are stored as bits in bit-addressable DATA memory which is available in all 8051 derivatives. Volatile directive declares volatile variable - variable which is modified by some interrupt or hardware. Absolute directive declares variable on top of another variable (*AbsVar absolute RecordVariable.Field* is also possible) or at some absolute address. Boolean variables can not be passed by reference (8051 has no instruction to reference bit variable by address) and can not be passed as parameter in re-entrant procedures. In such cases you can use system type *ByteBool* which occupies 1 byte. BitAddressable directive declares variable which will be placed in DATA address space from $20 to $2F - you can access individual bits of such (8-bit) bit-addressable variable with *BitAddressableVar.n* where n is 0 to 7. Data is always stored in little endian format.

Find some examples in **Appendix F**.

You can also declare variables in units with directive *absolute Forward* which means some (from the unit) unknown memory address. Example:

```
Unit I2C;

Interface

Var Ack: Boolean;
    SDA: Boolean absolute Forward;
    SCL: Boolean absolute Forward;
```

A main program which uses this unit declares these absolute Forward variables at correct address.

```
Program Test;

Uses I2C;

Var  I2C.SCL: Boolean absolute P3.4;
     I2C.SDA: Boolean absolute P3.5;
```

4 Procedures

4.1 System procedures

Assign

```
Procedure Assign (Var F: File; ReadFunction: Function; WriteProc: Procedure);
```

Procedure *Assign* assigns read function and write procedure to file variable *F*. Either *ReadFunction* or *WriteProc* can be omitted. Read function must be a non-reentrant function with no parameters which returns *Char* or *Byte* result (result must be returned in register A - default for Turbo51 pascal functions) and MUST preserve registers `R2, R3, R4, R5, R8, R9`. *WriteProc* must be a non-reentrant procedure with no parameters and MUST preserve registers `R2, R3, R6, R7`. Character to write is passed to procedure in register A. If the *WriteProc* is written in pascal then it must first save character to some local storage (short *asm* statement at the beginning of procedure).

BlockRead

```
Procedure BlockRead (Var F: File; Var Buffer; Count: Word);
```

Procedure *BlockRead* reads *Count* bytes from file *F* to Buffer. Files are read by the *ReadFunction* that is assigned to file *F*.

BlockWrite

```
Procedure BlockWrite (Var F: File; Var Buffer; Count: Word);
```

Procedure *BlockWrite* writes *Count* bytes from Buffer to file *F*. Bytes are written by the *WriteProcedure* that is assigned to file *F*.

Break

```
Procedure Break;
```

Break jumps to the statement following the end of the current loop statement. The code between the *Break* call and the end of the loop statement is skipped. This can be used with *Fo*r, *Repeat* and *While* statements.

Change

```
Procedure Change (S: TSetOfElement; Element: TOrdinalType);
```

Change changes inclusion of *Element* in the set *S* (If element is included in the set the procedure performs *Exclude* and *Include* otherwise).

Continue

```
Procedure Continue;
```

Continue jumps to the end of the current loop statement. The code between the *Continue* call and the end of the loop statement is skipped. This can be used with *For*, *Repeat* and *While* statements.

Dec

```
Procedure Dec (Var X: OrdinalType);
Procedure Dec (Var X: OrdinalType; Decrement: Longint);
```

Dec decrements the value of *X* with *Decrement*. If *Decrement* isn't specified, then 1 is taken as a default.

Delete

```
Procedure Delete (Var S: String; Index: Byte; Count: Byte);
```

Delete deletes *Count* characters from string *S*, starting at position *Index*. All characters after the deleted characters are shifted Count positions to the left, and the length of the string is adjusted.

Dispose

```
Procedure Dispose (P: Pointer);
Procedure Dispose (P: TypedPointer; Destruct: Procedure);
```

The first form *Dispose* releases the memory allocated with a call to *New*. The released memory is returned to the heap. The second form of *Dispose* accepts as a first parameter a pointer to an object type, and as a second parameter the name of a destructor of this object. The destructor will be called, and the memory allocated for the object will be freed.

Exclude

```
Procedure Exclude (S: TSetOfElement; Element: TOrdinalType);
```

Exclude excludes *Element* from the set *S*.

Exit

```
Procedure Exit;
```

Exit exits the current procedure or function and returns control to the calling routine.

ExitBlock

```
Procedure ExitBlock;
```

ExitBlock exits the current begin-end block and returns control to the statement after this begin-end block.

Fail

```
Procedure Fail;
```

Fail exits the constructor with *nil* value.

FillChar

```
Procedure Fillchar (Var Mem; Count: Word; Value: Char);
```

Fillchar fills the memory starting at *Mem* with *Count* characters with value equal to *Value*.

FreeMem

```
Procedure FreeMem (Var Ptr: Pointer; Count: Word);
```

FreeMem releases the memory occupied by the pointer *Ptr*, of size *Count* (in bytes), and returns it to the heap. *Ptr* should point to the memory allocated to a dynamic variable with procedure *GetMem*.

GetMem

```
Procedure GetMem (Var Ptr: Pointer; Size: Word);
```

GetMem reserves *Size* bytes memory on the heap, and returns a pointer to this memory in *Ptr*. If no more memory is available, *nil* is returned.

Halt

```
Procedure Halt;
```

Halt generates code for endless loop (i.e. jump to itself).

Inc

```
Procedure Inc (Var X: OrdinalType);
Procedure Inc (Var X: OrdinalType; Increment: Longint);
```

Inc increments the value of *X* with *Increment*. If *Increment* isn't specified, then 1 is taken as a default.

Include

```
Procedure Include (S: TSetOfElement; Element: TOrdinalType);
```

Include includes *Element* to the set *S*.

Insert

```
Procedure Insert (Const Source: String; Var DestStr: String; Index: Byte);
```

Insert inserts string *Source* in string *DestStr*, at position *Index*, shifting all characters after *Index* to the right. The resulting string is truncated at 255 characters, if needed.

Mark

```
Procedure Mark (Var Ptr: Pointer);
```

Mark copies the current heap-pointer *HeapPtr* to *Ptr*.

Move

```
Procedure Move (Var Source, Dest; Count: Word);
```

Move moves *Count* bytes from *Source* to *Dest*.

New

```
Procedure New (Var Ptr: Pointer);
Procedure New (Var Ptr: PObject; Constructor);
```

New allocates a new instance of the type pointed to by *Ptr*, and puts the address in *Ptr*. If *Ptr* is a pointer to an object, then it is possible to specify the name of the *constructor* with which the instance will be created.

Randomize

```
Procedure Randomize;
```

Randomize initializes the random number generator of Turbo51, by giving a value to *RandSeed,* calculated with the system clock.

Read

`Procedure Read ([Var F: File/Text; ] V1 [, V2, ... , Vn]);`

Read reads one or more values from the file assigned by variable *F* (see *Assign*), and stores the result in the variables *V1*, *V2*, ... If no file variable *F* is specified, then standard input is read. *F* can be one of the following types:

File of ..:	If *F* is a typed file, then each of the variables must be of the type specified in the declaration of *F*.
Text:	If *F* is a text file (such as the standard input), the characters read from input will be converted according to the specified variables (for example as an integer number).

Readln

`Procedure Readln ([Var F: File, ] V1 [, V2, ... , Vn]);`

Readln works like the procedure *Read* described above, but can only be used for text files. The values of the specified variables are read from input and stored as *V1*, *V2*, ... After that it goes to the next line in the file (defined by the line feed character (`#10`)). If no file *F* is specified, then standard input is read.

Release

`Procedure Release (Ptr: Pointer);`

Release sets the top of the heap to the location pointed to by *Ptr*. All memory at a location higher than *Ptr* is marked empty.

Str

`Procedure Str (Var X[: NumPlaces[:Decimals]]; Var Str: String);`

Str returns a string which represents the value of *X*. *X* can be any numerical type. The optional *NumPlaces* and *Decimals* specifiers control the formatting of the string.

Val

`Procedure Val (Const Str: String; Var V; Var ErrorCode: Integer);`

Val converts the value represented in the string *Str* to a numerical value, and stores this value in the variable *V*, which can be of type *Longint* or *Real*. If the conversion isn't successful, then the parameter *ErrorCode* contains the index of the character in *Str* which prevented the conversion. The string *Str* isn't allowed to contain spaces.

Write

`Procedure Write ([Var F: File, ] V1 [, V2, ... , Vn]);`

Write writes the contents of the variables *V1*, *V2* etc. to the file *F*. *F* can be a typed file, or a text file. If *F* is a typed file, then the variables *V1*, *V2* etc. must be of the same type as the type in the declaration of *F*. Untyped files are not allowed. If the parameter *F* is omitted, standard output is assumed (system file variable Output which is alias of text file *SystemIO*). If *F* is of type *Text*, then the necessary conversions are done such that the output of the variables is in character format. This conversion is done for all numerical types. Strings are printed exactly as they are in memory, as well as *PChar* types. The format of the numerical conversions can be influenced through the following modifiers: *OutputVariable: NumChars [: Decimals]*. This will print the value of *OutputVariable* with a minimum of *NumChars* characters, from which *Decimals* are reserved for the decimals. If the number cannot be represented with *NumChars* characters, *NumChars* will be increased, until the representation fits. If the representation requires less than *NumChars* characters then the output is filled up with spaces, to the left of the generated string, thus resulting in a right-aligned representation. If no formatting is specified, then the number is written using its natural length, with nothing in front of it if it's positive, and a minus sign if it's negative. Real numbers are, by default, written in scientific notation.

Writeln

```
Procedure Writeln ([Var F: File, ] V1 [, V2, ... , Vn]);
```

Writeln does the same as *Write* for text files, and writes a Carriage Return - LineFeed character pair (`#13#10`) after that. If the parameter *F* is omitted, standard output is assumed (system text variable Output which is alias of text file *SystemIO*). If no variables are specified, a Carriage Return - LineFeed character pair is written.

4.2 System functions

Abs

```
Function Abs (X: Integer): Integer;
Function Abs (X: Real): Real;
```

Abs returns the absolute value of a variable *X*. The result of the function has the same type as its argument, which can be Integer or Real.

Addr

```
Function Addr (X: T_DATA_Variable): ShortPtr;
Function Addr (X: T_XDATA_Variable): Pointer;
Function Addr (X: TProcedure): CodePointer;
```

Addr returns a pointer to its argument, which can be any type including procedure or function. If argument is in DATA segment the result is of type *ShortPtr*, if argument is in XDATA segment the result is of type *Pointer* and if argument is in CODE segment (typed constant, function, procedure, static method) the result is of type *CodePointer*. The returned pointer isn't typed. Similar result can be obtained by the *@ operator* which returns a typed pointer.

ArcTan

```
Function Arctan (X: Real): Real;
```

Arctan returns the Arctangent of *X*. The resulting angle is in radians.

Assigned

```
Function Assigned (P: Pointer): Boolean;
```

Assigned returns *True* if *P* is non-*nil* and returns *False* otherwise. *P* can be any pointer or procedural variable.

Bcd

```
Function Bcd (D: Byte): Byte;
```

Bcd returns binary coded decimal representation of *D*.

Chr

```
Function Chr (X: Byte): Char;
```

Chr returns the character which has ASCII value *X*.

Concat

```
Function Concat (S1, S2 [,S3, ... ,Sn]): String;
```

Concat concatenates the strings *S1*, *S2* etc. to one long string. The resulting string is truncated at a length of 255 bytes. The same operation can be performed with the + operation. Function *Concat* needs XDATA memory.

Copy

```
Function Copy (Const S: String; Index: Byte; Count: Byte): String;
```

Copy returns a string which is a copy if the *Count* characters in *S*, starting at position *Index*. If *Count* is larger than the length of the string *S*, the result is truncated. If *Index* is larger than the length of the string *S*, then an empty string is returned. Function *Copy* needs XDATA memory.

Cos

```
Function Cos (X: Real): Real;
```

Cos returns the cosine of *X*, where *X* is an angle in radians.

Exp

```
Function Exp (Var X: Real): Real;
```

Exp returns the exponent of *X*, i.e. the number *e* to the power *X*.

Frac

```
Function Frac (X: Real): Real;
```

Frac returns the fractional part of a floating point number in *X*.

Hi

```
Function Hi (X: Word): Byte;
Function Hi (X: Pointer): Byte;
```

Hi returns the high byte of word or pointer in *X*.

High

```
Function High (TOrdinalType): TOrdinalTypeElement;
Function High (X: TOrdinalType): TOrdinalTypeElement;
Function High (X: TArray): TArrayIndex;
Function High (X: TOpenArray): Integer;
```

The return value of *High* depends on it's argument:

1. If the argument is an ordinal type, *High* returns the highest value in the range of the given ordinal type.
2. If the argument is an array type or an array type variable then *High* returns the highest possible value of it's index.
3. If the argument is an open array identifier in a function or procedure, then *High* returns the highest index of the array, as if the array has a zero-based index. The return type is always the same type as the type of the argument (or type of index in arrays).

Int

```
Function Int (X: Real): Real;
```

Int returns the integer part of a floating point number in *X*. The result is *Real*, i.e. a floating point number.

Length

```
Function Length (S: String): Byte;
```

Length returns the length of the string *S*. If the strings *S* is empty, 0 is returned. Note: The length of the string *S* is stored in *S [0]*.

Ln

```
Function Ln (X: Real): Real;
```

Ln returns the natural logarithm of the *Real* parameter *X*. *X* must be positive.

Lo

```
Function Lo (X: Word): Byte;
Function Lo (X: Pointer): Byte;
```

Lo returns the low byte of *word* or *pointer* in *X*.

Low

```
Function Low (TOrdinalType): TOrdinalTypeElement;
Function Low (X: TOrdinalType): TOrdinalTypeElement;
Function Low (X: TArray): TArrayIndex;
Function Low (X: TOpenArray): Integer;
```

The return value of *Low* depends on it's argument:

1. If the argument is an ordinal type, *Low* returns the lowest value in the range of the given ordinal type.
2. If the argument is an array type or an array type variable then *Low* returns the lowest possible value of it's index.
3. If the argument is an open array identifier in a function or procedure, then *Low* returns the lowest index of the array which is 0. The return type is always the same type as the type of the argument (or type of index in arrays).

MaxAvail

```
Function MaxAvail: Word;
```

MaxAvail returns the size in bytes of the biggest free memory block in the heap.

MemAvail

```
Function MemAvail: Word;
```

MemAvail returns the size in bytes of all free memory in the heap.

New

```
Function New (PType);
Function New (PObjectType; Constructor);
```

New returns address of allocated memory for a new instance of the type *PType*. If *PType* is a pointer to an object type, then it is possible to specify the name of the *constructor* with which the instance will be created.

Odd

```
Function Odd (X: Longint): Boolean;
```

Odd returns *True* if *X* is odd, or *False* otherwise.

Ofs

```
Function Ofs (TRecord.Field): Longint;
```

Ofs returns offset of *Field* in record type *TRecord*.

Ord

```
Function Ord (X: TOrdinalType): Longint;
```

Ord returns the ordinal value of a ordinal-type variable *X*.

Pi

```
Function Pi: Real;
```

Pi returns the value of π (3.1415926535897932385).

Pos

```
Function Pos (Const Substr, Str: String): Byte;
```

Pos returns the index of *Substr* in *Str*, if *Str* contains *Substr*. In case *Substr* isn't found, 0 is returned. The search is case-sensitive.

Pred

```
Function Pred (X: TOrdinalType): TOrdinalType;
```

Pred returns the element that precedes the element that was passed to it.

Random

```
Function Random (L: Longint): Longint;
Function Random: Real;
```

Random returns a random number larger or equal to 0 and strictly less than *L*. If the argument *L* is omitted, a *Real* number between 0 and 1 is returned. (0 included, 1 excluded).

Round

```
Function Round (X: Real): Longint;
```

Round rounds *X* to the closest integer, which may be bigger or smaller than *X*.

Sin

```
Function Sin (X: Real): Real;
```

Sin returns the sine of its argument *X*, where *X* is an angle in radians.

SizeOf

```
Function SizeOf (TAnyType): Longint;
Function SizeOf (X: TAnyType): Longint;
Function SizeOf (TRecord.Field): Longint;
```

SizeOf returns the size in bytes of any variable, type or record field.

Sqr

```
Function Sqr (X: Real): Real;
```

Sqr returns the square of its argument *X*.

Sqrt

```
Function Sqrt (X: Real): Real;
```

Sqrt returns the square root of its argument *X*, which must be positive.

Succ

```
Function Succ (X: TOrdinalType): TOrdinalType;
```

Succ returns the element that succeeds the element that was passed to it.

Swap

```
Function Swap (X: Word): Word;
```

Swap swaps the high and low order bytes of *X*.

SwapWord

```
Function SwapWord (X: LongInt): LongInt;
```

SwapWord swaps the high and low order words of *X*.

Trunc

```
Function Trunc (X: Real): Longint;
```

Trunc returns the integer part of *X*, which is always smaller than (or equal to) *X* in absolute value.

TypeOf

```
Function TypeOf (TObjectType): Pointer;
```

TypeOf returns the address of the VMT of the *TObjectType*.

UpCase

```
Function Upcase (C: Char): Char;
```

UpCase returns the uppercase version of its argument *C*.

4.3 Assembler procedures

In **Appendix G** you can see some examples of procedures written entirely in 8051 assembly language. At the end of each procedure Turbo51 adds only `RET` instruction (or `RETI` in interrupt procedure). Turbo51 automatically removes `RET` instruction at the end of procedure if it finds out that it will not be reached. If for some reason `RET` instruction is not removed and you don't want it you can use the **`$NoReturn`** compiler directive inside assembler procedure to prevent generating `RET` instruction. You can easily pass parameters by value (Turbo51 automatically creates static variables for value storage), by reference (Turbo51 automatically creates static variables for pointer storage) or you can pass values in registers. Procedure's parameters can be accessed as local variables with *Procedure.Parameter*. See also assembler statement (5.1).

4.4 Inline procedures

Procedures (and functions) that are declared with the Inline directive are copied to the places where they are called. This has the effect that there is no actual procedure (or function) call, the code of the procedure is just copied to where the procedure is needed, this results in faster execution speed if the procedure or function is used a lot but but usually means also larger code size. You can override this behaviour with the **`$InlineCode`** directive. When set to *Off* (default is *On*) the compiler will generate normal calls to inline procedures.

Find an example in **Appendix H**.

4.5 Absolute procedures

You can force placing a procedure at absolute address with the *absolute* directive. This way you can also reserve some bytes at fixed addresses in code segment. There is no need to call procedures at absolute addresses, linker will place them where they should be.

See some examples in **Appendix I**.

4.6 Interrupts

Interrupts are procedures declared with the *Interrupt* directive and interrupt address. In this example *Timer0* is a constant defined in the System unit. For any procedure we can optionally define register bank to be used in this procedure by *Using* and number of bank (0 to 3) or we can define bank independent procedure with *UsingAny*. Such procedure can be called from any interrupt.

Warning: Make sure that all variables that might be changed in the interrupt procedure are marked with the *Volatile* directive. This will tell the compiler that their value can be modified outside of current program flow so many optimizations on these variables will not be performed since their value is not known. Do not place in an interrupt routine time consuming operations like floating point operations, file I/O, string manipulations, large memory moves, etc.

Find an example of interrupt declaration in **Appendix J**.

5 Assembler

5.1 Assembler statement

A Turbo51 assembler statement is very similar to 8051 assembler. You can use all instructions from the 8051 (MCS-51) instruction set. Labels starting with “@” don't have to be declared. You don't have to preserve any register and don't assume anything about register content before assembler statement. Byte variables *AR0* to *AR7* are direct locations for registers `R0` to `R7` (addresses from $00 to $1F, depending on the active register bank). To access an identifier which name is also name of a register place “&” before identifier name (example: use *&R0* to access identifier *R0* and not register `R0`). Procedure's parameters can be accessed as local variables with *Procedure.Parameter*.

See also assembler procedures (4.3). Find an example in **Appendix K**.

Additional notes:

`DB`	Use DB to define byte.
`DW`	Use DW to define word.
`DD`	Use DD to define double word (LongInt).
`OR`	Use OR for logical or operation.
`AND`	Use AND for logical and operation.
`XOR`	Use XOR for logical xor operation.
`NOT`	Use NOT for logical not operation.
`MOD`	Use MOD for integer division modulus.
`SHR`	Use SHR for right shift.
`SHL`	Use SHL for left shift.
`LOW (Word)`	Use LOW to access low byte of word.
`HIGH (Word)`	Use HIGH to access high byte of word.
`SWAP (Word)`	Use SWAP to swap low and high byte of word.
Arithmetic functions	Use +, -, *, / for integer arithmetic operations.
`Procedure.Parameter`	Use *Procedure.Parameter* to access called procedure's parameters.
`RecordType.Field`	Use *RecordType.Field* to get the offset of *field* in record.

`VMTADDRESS TObjectType`	Use *VMTADDRESS* to get the address of Virtual Method Table of *TObjectType*.
`VMTOFFSET TObjectType.VirtualMethod`	Use *VMTOFFSET* to get the offset of *TObjectType.VirtualMethod* in VMT.
`VMTADDRESSOFFSET TObjectType`	Use *VMTADDRESSOFFSET* to get the offset in object of the address of Virtual Method Table.
`DMTADDRESS TObjectType`	Use *DMTADDRESS* to get the address of Dynamic Method Table of *TObjectType*.
`DMTINDEX TObjectType.DynamicMethod`	Use *DMTINDEX* to get the index of *TObjectType.DynamicMethod*.

For reentrant procedures and functions the following symbols are defined:

`@LOCALS`	returns offset of local variables on XDATA stack.
`@PARAMS`	returns offset of parameters on XDATA stack.
`@RESULT`	returns offset of result variable on XDATA stack.

5.2 Compiler internals

If you are interested in Turbo Pascal internals and would like to see the source code of some popular commercial Pascal compiler then check Turbo Pascal Compiler Written in Turbo Pascal.

5.2.1 General

Data storage

All variables and typed constants are stored in little endian format.

Boolean variables

Boolean variables are stored as bits in bit-addressable DATA memory which is available in all 8051 derivatives. Boolean variables can not be passed by reference (8051 has no instruction to reference bit variable by address) and can not be passed as parameter in re-entrant procedures. In such cases you can use system type *ByteBool* which occupies 1 byte.

Global variables

Global variables are placed in default memory type for global variables which can be set with compiler directive **`$MG MemoryType`** (*`DATA`*, *`IDATA`* or *`XDATA`*) and defaults to *`DATA`*. This memory type can be overridden for each variable declaration.

Local variables

All local variables in normal (non-reentrant) procedures and functions are static. They are placed like global variables but are accessible only in local scope. Default memory type for local variables can be set with compiler directive **`$ML MemoryType`** (*`DATA`*, *`IDATA`* or *`XDATA`*) and defaults to *`DATA`*. This memory type can be overridden for each variable declaration.

Parameters

All parameters in normal (non-reentrant) procedures and functions are stored as local variables and are static. Default memory type for parameters can be set with compiler directive **`$MG MemoryType`** (*`DATA`*, *`IDATA`* or *`XDATA`*) and defaults to *`DATA.`* This memory type can be overridden for each parameter declaration.

Register usage

Turbo51 internally uses two register sets: R5R4R3R2 and R9R8R7R6. R8 and R9 are ordinary DATA variables declared in System unit. 8-bit data is stored in R2 (R6), 16-bit data is stored in R3R2 (R7R6) and 32-bit data uses whole set.

XDATA Stack

When XDATA memory is present Turbo51 creates there a stack. *XSP* (Pointer declared in System unit) points to to the top of stack.

Calls to normal (non-reentrant) procedures and functions

For normal, non-reentrant procedures all parameters are stored as local variables. Caller passes data by storing it to local memory for parameters and calls procedure. Functions return simple result in either ACC or first register set (R5R4R3R2). String functions return pointer to result string in R0 (if it is in DATA or IDATA memory) or in DPTR otherwise (result is in CODE or XDATA memory). Currently this is the only supported calling convention.

In most cases there is no need for reentrant procedures. This avoids using XDATA stack and greatly simplifies generated code. Try to avoid reentrant procedures unless they are really needed.

5.2.2 Reentrant procedures

For reentrant procedures all parameters are pushed on XDATA stack. Functions return simple result in first register set (R5R4R3R2). For functions which return a String result call must reserve space in XDATA memory and push its address (see Fig. 3). Before a call to the reentrant procedure is made the following is pushed on the XDATA stack:

- Address for String result (for functions which return String)
- All parameters in order in which they were declared
- *XBP* of calling local procedure

Called procedure on entry:

- pushes *XBP*
- sets *XBP* to point to the top of pushed parameters
- reserves space for local variables (increases *XSP* accordingly)

On exit called procedure pops saved *XBP* and removes all pushed parameters from XDATA stack. XDATA stack during reentrant procedure call looks like Fig. 3.

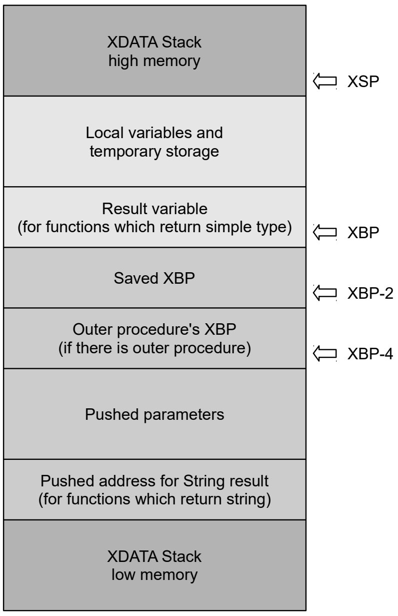

Fig. 3: Reentrant procedures - Use of external memory

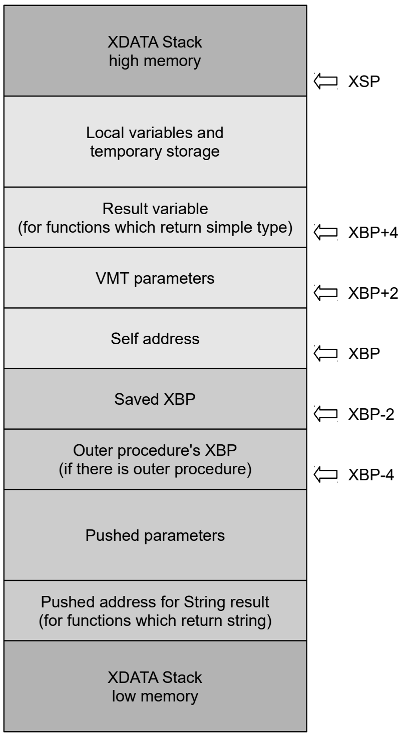

Fig. 4: Methods -Use of external memory

5.2.3 Methods

Methods are always reentrant. Before a call to the method is made the following is pushed on the XDATA stack:

- Address for String result (for functions which return String)
- All parameters in order in which they were declared

On call to static method the following parameters must be in registers:

- `R3R2`: Self address

On call to virtual method the following parameters must be in registers:

- `DPTR`: Self address (will be placed in `R3R2` by system routine for virtual method call)
- `R2` (or `R3R2`): Offset of VMT address
- `R0` (or `R1R0`): Offset of method address
- `R5R4`: VMT parameter

On call to dynamic method the following parameters must be in registers:

- `DPTR`*:* Self address (will be placed in `R3R2` by system routine for dynamic method call)
- `R2` (`R3R2`): Offset of DMT address
- `R1`: Dynamic method index
- `R5R4`: VMT parameter

On call to constructor method the following parameters must be in registers:

- `R3R2`: Self address (If *nil*, constructor call is via *New*)
- `R5R4`: VMT parameter (address of VMT - normal call, $0000 means static call -no initialization of VMT address in *Self*)
- Returns *Self* (address of allocated object) in `R3R2`

On call to destructor method the following parameters must be in registers:

- `R3R2`: Self address

- `R4`: VMT parameter ($00: normal *destructor* call, $01: call via *Dispose*)

Called method on entry:

- pushes *XBP*
- sets *XBP* to point to the top of pushed parameters
- pushes *Self* parameter which was passed in `R3R2`
- pushes VMT parameter which was passed in `R5R4`
- reserves space for local variables (increases *XSP* accordingly)

On exit called method pops saved *XBP* and removes all pushed parameters from XDATA stack. XDATA stack during called method looks like Fig 4.

Appendix A – System unit:

```
Unit System;

Interface

Const
  BELL = $07;
  BS   = $08;
  TAB  = $09;
  LF   = $0A;
  CR   = $0D;
  EOF  = $1A;
  ESC  = $1B;
  DEL  = $7F;

{ Interrupt addresses valid for all 8051 microcontrollers }

  External0 = $0003;
  Timer0    = $000B;
  External1 = $0013;
  Timer1    = $001B;
  Serial    = $0023;

Type
  TDeviceWriteProcedure = Procedure;
  TDeviceReadFunction = Function: Char;
  TFileRecord = Record
    WriteProcedure: TDeviceWriteProcedure;
    ReadFunction:   TDeviceReadFunction;
    end;
Var
{ Direct access to 8051 registers R0 to R7, exact address is bank dependent and will
be set by the linker }

  AR0:      Byte absolute 0;
  AR1:      Byte absolute 1;
  AR2:      Byte absolute 2;
  AR3:      Byte absolute 3;
  AR4:      Byte absolute 4;
  AR5:      Byte absolute 5;
  AR6:      Byte absolute 6;
  AR7:      Byte absolute 7;

{ SFRs present in all 8051 microcontrollers }

  P0:       Byte absolute $80; Volatile;
  SP:       Byte absolute $81; Volatile;
  DPL:      Byte absolute $82;
  DPH:      Byte absolute $83;
  PCON:     Byte absolute $87; Volatile;
  TCON:     Byte absolute $88; Volatile;
  TMOD:     Byte absolute $89; Volatile;
  TL0:      Byte absolute $8A; Volatile;
  TL1:      Byte absolute $8B; Volatile;
  TH0:      Byte absolute $8C; Volatile;
  TH1:      Byte absolute $8D; Volatile;
  P1:       Byte absolute $90; Volatile;
  SCON:     Byte absolute $98; Volatile;
  SBUF:     Byte absolute $99; Volatile;
  P2:       Byte absolute $A0; Volatile;
  IE:       Byte absolute $A8; Volatile;
  P3:       Byte absolute $B0; Volatile;
  IP:       Byte absolute $B8; Volatile;
  PSW:      Byte absolute $D0; Volatile;
  ACC:      Byte absolute $E0;
  B:        Byte absolute $F0;

  DPTR:     Pointer absolute $82;

{ TCON }
  TF1:      Boolean absolute TCON.7;
  TR1:      Boolean absolute TCON.6;
  TF0:      Boolean absolute TCON.5;
  TR0:      Boolean absolute TCON.4;
  IE1:      Boolean absolute TCON.3;
  IT1:      Boolean absolute TCON.2;
  IE0:      Boolean absolute TCON.1;
  IT0:      Boolean absolute TCON.0;

{ SCON }
  SM0:      Boolean absolute SCON.7;
  SM1:      Boolean absolute SCON.6;
  SM2:      Boolean absolute SCON.5;
  REN:      Boolean absolute SCON.4;
  TB8:      Boolean absolute SCON.3;
  RB8:      Boolean absolute SCON.2;
  TI:       Boolean absolute SCON.1;
  RI:       Boolean absolute SCON.0;

{ IE }
  EA:       Boolean absolute IE.7;
  ES:       Boolean absolute IE.4;
  ET1:      Boolean absolute IE.3;
  EX1:      Boolean absolute IE.2;
  ET0:      Boolean absolute IE.1;
  EX0:      Boolean absolute IE.0;

{ P3 }
  RD:       Boolean absolute P3.7;
  WR:       Boolean absolute P3.6;
  T1:       Boolean absolute P3.5;
  T0:       Boolean absolute P3.4;
  INT1:     Boolean absolute P3.3;
  INT0:     Boolean absolute P3.2;
  TXD:      Boolean absolute P3.1;
  RXD:      Boolean absolute P3.0;

{ IP}
  PS:       Boolean absolute IP.4;
  PT1:      Boolean absolute IP.3;
  PX1:      Boolean absolute IP.2;
  PT0:      Boolean absolute IP.1;
  PX0:      Boolean absolute IP.0;
```

```
{ PSW }
  CY:       Boolean absolute PSW.7;
  AC:       Boolean absolute PSW.6;
  F0:       Boolean absolute PSW.5;
  RS1:      Boolean absolute PSW.4;
  RS0:      Boolean absolute PSW.3;
  OV:       Boolean absolute PSW.2;
  P:        Boolean absolute PSW.0;

  MemCODE:  Array [$0000..$FFFF] of Byte CODE  absolute $0000;
  MemDATA:  Array [  $00..  $FF] of Byte DATA  absolute   $00;  { Present in all 8051
microcontrollers, addresses from $80 and above access SFRs }
  MemIDATA: Array [  $00..  $FF] of Byte IDATA absolute   $00;  { IDATA memory from
$80..$FF is not present in all 8051 microcontrollers }
  MemXDATA: Array [$0000..$FFFF] of Byte XDATA absolute $0000;  { Not present in all
8051 microcontrollers, usually added externally }

Var
  XDATA_StackStart: Word XDATA;  { Used for XSP and XBP initialization }
  StackStart:   Byte DATA;       { Used for SP initialization }
  R8, R9:       Byte DATA;       { Used for LongInt set 1 }
  TempRegister: Byte DATA;

   { Used for recursion stack and local variables in XDATA }

     XSP, XBP:     Pointer DATA;

   { Used for heap management }

     HeapOrg,
     HeapPtr,
     HeapEnd:      Pointer DATA;
     FreeList:     Pointer XDATA;
     HeapError:    Procedure DATA;

     RandomSeed:   LongInt DATA;    { Used for random numbers }

   { Used for file I/O }

     CurrentIO:    File DATA;
     SystemIO:     Text DATA;       { Used for Read/Readln/Write/Writeln }
     Input:        Text absolute SystemIO;
     Output:       Text absolute SystemIO;

   { Used for sysReadCharFromCurrentDevice }

     LastCharacterBuffer:      Char;
     LastCharacterBufferValid: Boolean;

   { Used for some arithmetic functions }

     Overflow: Boolean;
     ResultSign: Boolean;
     TempBool0: Boolean;
     TempBool1: Boolean;
     TempWord:  Word DATA;
     TempByte0: Byte DATA;
     TempByte1: Byte DATA;
```

```
      TempByte2: Byte DATA;
      TempByte3: Byte DATA;
      TempByte4: Byte DATA;
      TempByte5: Byte DATA;
      TempByte6: Byte DATA;
      TempByte7: Byte DATA;
      TempByte8: Byte DATA;
      TempByte9: Byte DATA;

Var RealSigns: Byte DATA;
    RealResult: LongInt DATA;
    RealResultCarry: Byte DATA;

Const Pi_2:     Real = Pi / 2;
      Pi_24:    Real = Pi / 24;
      _Pi:      Real = Pi;
      _2Pi:     Real = 2 * Pi;
      _2_Pi:    Real = 2 / Pi;
      _0_5:     Real = 0.5;
      _1:       Real = 1;
      Sqrt2:    Real = Sqrt (2);
      _1_Sqrt2: Real = 1 / Sqrt (2);
      Ln2:      Real = Ln (2);
      Ln2_2:    Real = Ln (2) / 2;
```

Appendix B - simple example of a calculator using files:

```
Program Files;

// Should work on any 8051 microcontroller

Const
 Oscillator = 22118400;
 BaudRate   = 19200;
 BaudRateTimerValue = Byte (- Oscillator div 12 div 32 div BaudRate);

Var SerialPort: Text;
    Num1, Num2: LongInt;

Procedure WriteToSerialPort; Assembler;
Asm
  CLR   TI
  MOV   SBUF, A
@WaitLoop:
  JNB   TI, @WaitLoop
end;

Function ReadFromSerialPort: Char;
Var ByteResult: Byte absolute Result;
begin
  While not RI do;
  RI := False;
  ByteResult := SBUF;

{ Echo character }

  Asm
    MOV    A, Result
    LCALL  WriteToSerialPort
  end;
  if ByteResult=13 then ByteResult:=10; // linefeed required as end of line
end;


Procedure Init;
begin
  TL1  := BaudRateTimerValue;
  TH1  := BaudRateTimerValue;
  TMOD := %00100001;    { Timer1: no GATE, 8 bit timer, autoreload }
  SCON := %01010000;    { Serial Mode 1, Enable Reception }
  TI   := True;         { Indicate TX ready }
  TR1  := True;         { Enable timer 1 }
end;

begin
  Init;
  Assign (SerialPort, ReadFromSerialPort, WriteToSerialPort);

  Writeln (SerialPort, 'Turbo51 IO file demo');
  Repeat
    Write   (SerialPort, 'Enter first number: ');
    Readln  (SerialPort, Num1);
    Write   (SerialPort, 'Enter second number: ');
    Readln  (SerialPort, Num2);
    Writeln (SerialPort, Num1, ' + ', Num2, ' = ', Num1 + Num2);
  until False;
end.
```

Appendix C - Example program of how to use objects:

```
Program OOP;

{$M $0000, $1000, $0000, $1000, 0}
Type
  TLocation = Object
                X, Y : Integer;
                Procedure Init (InitX, InitY: Word);
                Function  GetX: Word;
                Function  GetY: Word;
              end;

  TPoint = Object (TLocation)
             Visible:  ByteBool;
             Procedure Init (InitX, InitY: Word);
             Procedure Show;
             Procedure Hide;
             Function  IsVisible: byteBool;
             Procedure MoveTo (NewX, NewY: Word);
           end;

Const
 clBlack = 0;
 clGreen = 2;

Var Point: TPoint XDATA;
```

```
Procedure PutPixel (X, Y: Word; Color: Byte);
begin
// Code to draw pixel
end;

Procedure TLocation.Init (InitX, InitY: Word);
begin
  X := InitX;
  Y := InitY;
end;

Function TLocation.GetX: Word;
begin
  GetX := X;
end;

Function TLocation.GetY: Word;
begin
  GetY := Y;
end;

Procedure TPoint.Init (InitX, InitY: Word);
begin
  TLocation.Init (InitX, InitY);
  Visible := False;
end;

Procedure TPoint.Show;
begin
  Visible := True;
  PutPixel (X, Y, clGreen);
end;

Procedure TPoint.Hide;
begin
  Visible := False;
  PutPixel (X, Y, clBlack);
end;

Function TPoint.IsVisible: ByteBool;
begin
  IsVisible := Visible;
end;

Procedure TPoint.MoveTo (NewX, NewY: Word);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;

begin
  Point.Init (100, 50);    // Initial X,Y at 10, 50
  Point.Show;              // APoint turns itself on
  Point.MoveTo (120, 100); // APoint moves to 120, 100
  Point.Hide;              // APoint turns itself off
  With Point do
    begin
      Init (100, 50);    // Initial X, Y at 100, 50
      Show;              // APoint turns itself on
      MoveTo (120, 100); // APoint moves to 120, 100
      Hide;
    end;
end.
```

Appendix D - Examples of constant definitions:

```
Const
  SystemClock              = 22118400;
  ConversionClockValue     = (SystemClock div ConversionClock - 1) shl 3;
  PeriodicTimerValue       = - SystemClock div 12 div TimerIntsPerSecond;

  SampleFrequency_10       = SamplesPerBit * 11875;
  RDS_SampleRateTimerValue = - 10 * SystemClock div SampleFrequency_10;
  GroupTime                = 1040000 div 11875;
  SMB0CRValue              = (2 - SystemClock div 2 div SMBusClock) and $FF;
  BaudRateTimerValue1      = - SystemClock div 32 div BaudRateSerial1;

  BaudRateTimerValue_19200 = Word (- SystemClock div 32 div 19200);
  BaudRateTimerValue_38400 = Word (- SystemClock div 32 div 38400);

  BaudRateTimerValue: Array [$01..$02] of Word = (
    BaudRateTimerValue_19200,
    BaudRateTimerValue_38400);

  LedTime                  =  30;

  SampleTable_0_0_0: Array [0..SamplesPerBit - 1] of Word =
    ({$I Table_0_0_0.inc });

  VersionHi = $01;
  VersionLo = $00;

  Signature = $8051;     // Identify 8051 microcontroller

  Greeting = 'PASCAL 8051'#0;
  GreetingString: Array [0..Length (Greeting) - 1] of Char = Greeting;

  ManufacturerKeyData: Array [0..7] of Byte =
       ($DA, $FE, $02, $40, $03, $3D, $B5, $CA);

  BaudRateTimerValue = Word (- 22118400 div 32 div 9600);

  LedTime         =    30;
  RS485_Time      =     2;
  SetupTime       = 30000;
  NoKeyTime       =   100;

  DafaultRelayLongPulseTime  =   30000 div 32;
  DafaultLightPulseTime      = 1800000 div 32;
  DefaultRelayShortPulseTime =     384 div 32;
  DefaultRelayOffPulseTime   =    1500 div 32;

  Relay_ON  = LowLevel;
```

```
   Relay_OFF = HighLevel;

   MotorUp   = 1;
   MotorDown = 0;

   HexChar: Array [0..15] of Char = '0123456789ABCDEF';

   BlockStart    = $AA;
   BlockStartLNG = $CA;

   bpBlockStart         = 0;
   bpDestinationAddress = 1;
   bpcommand            = 2;
   bpParameter1         = 3;
   bpParameter2         = 4;
   bpSourceAddress      = 5;
   bpChecksum           = 6;

   bpParameter3         = 7;
   bpParameter4         = 8;

   LNG_ModuleID         = $A0;

   Cmd_Relay            = $21;
```

Appendix E - Examples of type declarations:

```
Type
    PDataWordX = ^TDataWord XDATA;
    TDataWord = Record
                  Case Byte of
                   0: (Word: Word);
                   1: (Byte0, Byte1: Byte);
                   2: (Bits: Set of 0..15);
                   3: (Pointer: Pointer);
                end;

    TGroupType = (Group_0A, Group_0B, Group_15A, Group_15B);

    PByte = ^Byte;
    PByteX = ^Byte XDATA;
    PPointerX = ^Pointer XDATA;
    TPS = Array [1..8] of Char;
    TRT_Text = Array [1..64] of Char;
    TRT_Flags = (rtToggleAB);
    TRT_FlagSet = Set of TRT_Flags;
    PRTX = ^TRT XDATA;
    TRT = Record
            Flags: TRT_FlagSet;
            RepeatNumber: LongInt;
            Text: TRT_Text;
          end;

    TAF = Array [0..NumberOf_AF_FrequenciesInDataSet - 1] of Byte;

    PEON_AF_X = ^TEON_AF XDATA;
    TEON_AF = Record
                Variant: LongInt;
```

```
              Case Byte of
                0: (AF_DataWord: Word);
                1: (AF_Data1: Byte; AF_Data2: Byte);
            end;

  TEEPROM_Data = Record
                  eeSignature:             TSignature;
                  eeDataSet:               Array [1..NumOfDS] of TDataSet;
                  eeEnd:                   Byte;
                end;
```

Appendix F - Some examples of variable declarations:

```
Var
   EthernetReset:                Boolean absolute P0.7;
   EthernetMode:                 Boolean absolute P0.6;

   TempString:                   String [24] XDATA;
   TempChecksum:                 Byte;
   TempByte2:                    Byte absolute TempChecksum;
   DelayTimer:                   Word XDATA; Volatile;
   SamplePulse:                  Boolean;
   InputSync:                    Byte; BitAddressable;
   VideoSync1:                   Boolean absolute InputSync.0;
   VideoSync2:                   Boolean absolute InputSync.1;
   LastPCPort:                   TActiveBuffer;

   RemoteTemperatureReadState:   TRemoteTemperatureReadState XDATA;
   TempRemoteTemperature,
   RemoteTemperature:            Array [1..4] of Word XDATA;
   RemoteTemperatureThreshold:   Word XDATA;

 {$IFDEF TEST }
   TempTimer:                    Word; Volatile;
   TxBuffer1:                    Array [0..15] of Byte IDATA;
 {$ENDIF }

   BroadcastReplyTimer_Serial0: Word IDATA; Volatile;

   UART:                         Array [1..4] of TUART XDATA;

   TX_Buffer_Serial0:            TExtendedGeneralPacket XDATA;
   TX_BufferArray_Serial0:       TBufferArray absolute TX_Buffer_Serial0;

   PRX_Buffer_Cmd_Message:       ^TCmd_Message XDATA absolute PRX_Buffer;

   UartData:                     Array [1..4] of TUartData IDATA;
   RX_Buffer_UART:               Array [1..4] of TExtendedGeneralPacket XDATA;

   EEPROM_Data:                  TEEPROM_Data XDATA absolute 0;
```

Appendix G - Example of assembler procedures:

```
Program AssemblerProcedures;

{ Useless program just to demonstrate assembler procedures }
```

```
Const DemoText       = 'Turbo51 assembler procedures demo';
      ZeroTerminated = 'Zero terminated';

      DemoString: String [Length (DemoText)] = DemoText;
      String0:    Array [0 .. Length (ZeroTerminated)] of Char = ZeroTerminated;

Var Number1, Number2, Result: Word;

Procedure Add; Assembler;
Asm
  MOV       A, Number1
  ADD       A, Number2
  MOV       Result, A
  MOV       A, Number1 + 1
  ADDC      A, Number2 + 1
  MOV       Result + 1, A
end;

Procedure Multiply; Assembler;
Asm
  MOV       R2, Number1
  MOV       R3, Number1 + 1
  MOV       R6, Number2
  MOV       R7, Number2 + 1

  MOV       A, R2
  MOV       B, R6
  MUL       AB
  XCH       A, R2
  XCH       A, R7
  XCH       A, B
  XCH       A, R7
  MUL       AB
  ADD       A, R7
  XCH       A, R3
  MOV       B, R6
  MUL       AB
  ADD       A, R3
  MOV       R3, A

  MOV       Result, R2
  MOV       Result + 1, R3
end;


Procedure CalculateXorValue (Num1, Num2: Word); Assembler;
Asm
  MOV       A, Num1
  XRL       A, Num2
  MOV       R2, A
  MOV       A, Num1 + 1
  XRL       A, Num2 + 1
  MOV       R3, A
end;

Procedure SwapWords (Var Num1, Num2: Word); Assembler;
Asm
  MOV       R0, Num1
  MOV       R1, Num2

  MOV       B, @R0
  MOV       A, @R1
  MOV       @R0, A
  MOV       @R1, B

  INC       R0
  INC       R1

  MOV       B, @R0
  MOV       A, @R1
  MOV       @R0, A
  MOV       @R1, B
end;

Procedure WriteString; Assembler;
Asm
//  Code to write string in code with address in DPTR
end;

Procedure WriteZeroTerminatedString; Assembler;
Asm
//  Code to write zero terminated string in code with address in DPTR
end;

Procedure WriteResult; Assembler;
Asm
//  Code to write number in Result variable
end;

begin
  Asm
    MOV       DPTR, #DemoString
    LCALL     WriteString

    MOV       DPTR, #String0
    LCALL     WriteZeroTerminatedString

    MOV       Number1,     #LOW  (200)
    MOV       Number1 + 1, #HIGH (200)
    MOV       Number2,     #LOW  (40)
    MOV       Number2 + 1, #HIGH (40)
    LCALL     Add
    LCALL     WriteResult

    MOV       Number1,     #LOW  (2000)
    MOV       Number1 + 1, #HIGH (2000)
    MOV       Number2,     #LOW  (45)
    MOV       Number2 + 1, #HIGH (45)
    LCALL     Multiply
    LCALL     WriteResult

    MOV       CalculateXorValue.Num1,     #LOW  ($1234)
    MOV       CalculateXorValue.Num1 + 1, #HIGH ($1234)
    MOV       CalculateXorValue.Num2,     #LOW  (10000)
    MOV       CalculateXorValue.Num2 + 1, #HIGH (10000)
    LCALL     CalculateXorValue
    MOV       Result,     R2
    MOV       Result + 1, R3
    LCALL     WriteResult

    MOV       Number1,     #LOW  (2000)
    MOV       Number1 + 1, #HIGH (2000)
    MOV       Number2,     #LOW  (45)
    MOV       Number2 + 1, #HIGH (45)
    MOV       SwapWords.Num1, #Number1
    MOV       SwapWords.Num2, #Number2
    LCALL     SwapWords
    LCALL     WriteResult
  end;
end.
```

Appendix H - Example of inline procedures:

```
{
    This file is part of the Turbo51 code examples.
    Copyright (C) 2008 by Igor Funa

    http://turbo51.com/

    This file is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
}

Program Example4;

{ Usless program just to demonstrate sets and inline procedures/functions }

Type
  TFlag = (fl0, fl1, fl2, fl3, fl4, fl5, fl6);
  TFlagsSet = Set of TFlag;

  TVariantRecord = Record
                     Case Byte of
                       0: (L: LongInt);
                       1: (W0, W1: Word);
                       2: (B0, B1, B2, B3: Byte);
                       3: (DataWord: Word; LocalFlags: Set of 0..7; Flags: TFlagsSet);
                       4: (IndividualBits: Set of 0..31);
                       5: (Ch0, Ch1, Ch2, Ch3: Char);
                   end;

Const
   InitialFlags = [fl0, fl4, fl5];
   TempFlags    = [fl0, fl1, fl3];

Var
  WatchdogClock: Boolean absolute P0.4;

  GlobalFlags: TFlagsSet;
  DataRecord1,
  DataRecord2: TVariantRecord;
  Character: Char;
  B1, B2: Byte;

Function UpcaseChar (Ch: Char): Char;
{$I InlineChar.inc }

Function InlineUpcaseChar (Ch: Char): Char; Inline;
{$I InlineChar.inc }

Procedure RestartWatchdog; Inline; Assembler;
Asm
  CPL  WatchdogClock;
end;

Procedure Multiply (Var Factor: Byte);
begin
  Factor := Factor * 10;
  If Factor >= 100 then Factor := 0;
end;

Procedure InlineMultiply (Var Factor: Byte); Inline;
begin
  Factor := Factor * 10;
  If Factor >= 100 then Factor := 0;
end;

begin
  GlobalFlags := InitialFlags;

  Include (GlobalFlags, fl4);
  Exclude (GlobalFlags, fl5);
  Change  (GlobalFlags, fl6);

  RestartWatchdog;

  DataRecord1.L := $12345678;
  With DataRecord2 do
    begin
      DataWord    := DataRecord1.W0 + DataRecord1.W1;
      LocalFlags := [3, 5, 6];
      Flags       := GlobalFlags;
    end;

  RestartWatchdog;

  Case fl6 in DataRecord2.Flags of
    True: DataRecord2.Flags := DataRecord2.Flags * TempFlags + [fl2, fl3];
    else  DataRecord2.Flags := TempFlags;
  end;

  RestartWatchdog;

  If 0 in DataRecord2.IndividualBits then With DataRecord2 do
    begin
      Include (IndividualBits, 4);
      Exclude (IndividualBits, 15);
      Change  (IndividualBits, 31);
    end;
```

```
{ Call to function UpcaseChar }

  Character := Chr (Ord ('a') + Random (Ord ('z') - Ord ('a') + 1));
  DataRecord1.Ch0 := UpcaseChar (Character);

{ Inline function InlineUpcaseChar }

  Character := Chr (Ord ('a') + Random (Ord ('z') - Ord ('a') + 1));
  DataRecord1.Ch1 := InlineUpcaseChar (Character);

{ Call to procedure Multiply }

  Multiply (DataRecord1.B1);

{ Inline procedure InlineMultiply }

  InlineMultiply (DataRecord1.B1);

{$InlineCode Off }

{ Normal call to Inline function InlineUpcaseChar }

  Character := Chr (Ord ('a') + Random (Ord ('z') - Ord ('a') + 1));
  DataRecord1.Ch1 := InlineUpcaseChar (Character);

{ Normal call to Inline procedure InlineMultiply }

  InlineMultiply (DataRecord1.B1);
end.
```

Appendix I - Examples of absolute procedures:

```
Program AbsoluteProcedures;

{ Usless program just to demonstrate procedures/functions at absolute addersses }

{ This procedure will be placed at code address $1000 and will occupy just one byte
(RET) }

Procedure MustBeFixed absolute $1000;
begin
end;


{ This procedure will also occupy just one byte at code address $0045 }

Procedure JustOneByte absolute $45; Assembler;
Asm
  DB    $00
{$NoReturn }    { Don't generate RET instruction }
end;


{ This procedure will be placed at code address $F000 }

Procedure Restart absolute $F000;
begin
```

```
  Asm
    LJMP   $0000
  end;
end;

begin
// no need to call procedures at absolute addresses, linker will just put them where
they should be
end.
```

Appendix J - Examples of interrupt procedures:

```
Program InterruptDemo;

Const
  Const1ms = - 22118400 div 12 div 1000;

Var
  RS485_TX: Boolean absolute P0.3;
  RX_Led:   Boolean absolute P0.4;
  TX_Led:   Boolean absolute P0.5;

  BlinkTimer:         Word; Volatile;
  KeyProcessingTimer: Word; Volatile;
  DelayTimer:         Word; Volatile;
  RS485_Timer:        Byte; Volatile;
  RX_LedTimer:        Byte; Volatile;
  TX_LedTimer:        Byte; Volatile;

Procedure TimerProc; Interrupt Timer0; Using 2; { 1 ms interrupt }
 begin
   TL0 :=  Lo (Const1ms);
   TH0 :=  Hi (Const1ms);

   Inc (BlinkTimer);
   Inc (KeyProcessingTimer);

   If DelayTimer <> 0 then Dec (DelayTimer);

   If RS485_Timer <> 0 then Dec (RS485_Timer) else RS485_TX := False;
   If RX_LedTimer <> 0 then Dec (RX_LedTimer) else RX_Led := False;
   If TX_LedTimer <> 0 then Dec (TX_LedTimer) else TX_Led := False;
 end;

begin
 { Some code }
end.
```

Appendix K - Examples of assembler statements:

```
Asm
    MOV      R2, UECP_RX_BufferReadPointer
    MOV      R3, UECP_RX_BufferReadPointer + 1
    MOV      R4, FrameCRCAddress
    MOV      R5, FrameCRCAddress + 1
  @1:
    MOV      DPL, R2
    MOV      DPH, R3
    MOVX     A, @DPTR
    INC      DPTR
    MOV      R2, DPL
    MOV      R3, DPH

    XRL      A, RX_CRC + 1
    MOV      B, #2
    MUL      AB
    ADD      A, #LOW  (CrcTable)
    MOV      DPL, A
    MOV      A, #HIGH (CrcTable)
    ADDC     A, B
    MOV      DPH, A

    CLR      A
    MOVC     A, @A+DPTR
    XRL      A, RX_CRC
    MOV      RX_CRC + 1,A
    MOV      A, #1
    MOVC     A, @A + DPTR
    MOV      RX_CRC, A

    MOV      A, R2
    XRL      A, R4
    JNZ      @1
    MOV      A, R3
    XRL      A, R5
    JNZ      @1

    MOV      DPL, R2
    MOV      DPH, R3    { DPTR points to CRC }
    INC      DPTR       { Move to next frame }
    INC      DPTR
    MOV      UECP_RX_BufferReadPointer, DPL
    MOV      UECP_RX_BufferReadPointer + 1, DPH

    MOV      A, RX_CRC
    XRL      A, #$FF
    XCH      A, RX_CRC + 1
    XRL      A, #$FF
    MOV      RX_CRC, A

    MOV      StoreData.CRC, A
    MOV      StoreData.ReadPointer, DPL
    MOV      StoreData.ReadPointer + 1, DPH
    LCALL    StoreData
end;
```