Turbo-51

Dokumentation

von Igor Funa

übersetzt von
Jürgen Rathlev

Inhaltsverzeichnis

1 Einführung....1
2 Allgemeines....2
2.1 Befehlszeilen-Syntax....2
2.2 Compiler-Schalter und -Direktiven....3
2.3 Speicherorganisation....5
2.4 System-Unit....6
2.5 Dateien....6
2.6 Objekte....6
3 Deklarationen....6
3.1 Konstanten....6
3.2 Typen....7
3.3 Variablen....7
4 Prozeduren....8
4.1 System-Prozeduren....8
4.2 System-Funktionen....12
4.3 Assembler-Prozeduren....17
4.4 Inline-Prozeduren....17
4.5 Absolute Prozeduren....17
4.6 Interrupts....17
5 Assembler....18
5.1 Assembler-Anweisung....18
5.2 Compiler-Interna....19
5.2.1 Allgemeines....19
5.2.2 Ablaufinvariante (reentrant) Prozeduren....20
5.2.3 Methoden....21
Anhang A – System-Unit:....23
Anhang B – einfaches Beispiel für die Ein- und Ausgabe von Daten:....26
Anhang C – Beispiel für ein Programm mit Objekten:....27
Anhang D – Beispiele für die Definition von Konstanten:....29
Anhang E – Beispiele für die Deklaration von Typen:....30
Anhang F – Beispiele für die Deklaration von Variablen:....31
Anhang G – Beispiele für Assembler-Prozeduren:....32
Anhang H – Beispiele für Inline-Prozeduren:....34
Anhang I – Beispiele für absolute Prozeduren:....36
Anhang J – Beispiele für Interrupt-Prozeduren:....37
Anhang K – Beispiele für Assembler-Anweisungen:....38

1 Einführung

Turbo51 von Igor Funa ist ein frei verfügbarer Pascal-Compiler für die 8051-Mikrocontroller-Familie. Wer Programme für diese Mikrocontroller entwickelt und das Programmieren in Pascal vorzieht, wird mit Turbo51 viel Freude haben.

Die wichtigsten Eigenschaften:	**Verwendete Optimierungen:**
• Win32-Konsolenanwendung • Schneller optimierender 1-Pass-Compiler • Syntax wie bei Turbo-Pascal-7 von Borland • Vollständige Fließkomma-Arithmetik • Gemischter Pascal und Assembler-Code möglich • Vollständige Verwendung der Register-Bänke • Erweiterte Mehrpass-Optimierung • Intelligenter Linker • Erzeugt kompakten hochwertigen Code • Ausgabeformate: BIN, HEX, OMF • Generierung von Assembler-Quelltext • Quelltext-basierte Fehlersuche mit erweiterter OMF-51-Objektdatei	• 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 wird als Freeware bereitgestellt. Es kann sowohl für Hobby-Projekte als auch für ernsthafte Anwendungen benutzt werden. Lesen Sie diese Dokumentation und Code-Beispiele, um sich mit der Syntax, den Möglichkeiten und den erzeugten Dateien vertraut zu machen. Das sollte für einen Einstieg in die 8051-Programmierung mit Trubo51 ausreichen. Bei Problemen steht der Autor stets für Fragen und Hilfestellungen zur Verfügung.

Wer sich schon mit der 8051-Assembler-Programmierung auskennt, kann Turbo51 zunächst als Compiler für den Assembler-Code benutzen und dann nach und nach Pascal-Anweisungen einfügen, um sich auch damit vertraut zu machen. Wer Erfahrungen mit Turbo-Pascal mitbringt, sollte mit kleinen Pascal-Programmen beginnen und sich dann den von Turbo51 erzeugten Assemblercode ansehen. So lernt man Schritt für Schritt die Assemblersprache und erhält auch einige Tipps, wie ein effektiver Code zu schreiben ist. Wie viel andere verbreitete C-Compiler für den 8051, erzeugt auch Turbo51 optimierten Code und unterstützt das OMF-Format für die quelltext-basierte Fehlersuche.

Turbo51 ist eine Befehlszeilen-Anwendung, d.h. es gibt keine grafische Benutzeroberfläche, keine Menüs u.ä. Man benötigt also einen Texteditor für das Schreiben der Programme und startet die Übersetzung über die Befehlszeile mit den passenden Optionen. Dies kann bei vielen Texteditoren als externe Anwendung integriert werden, so dass das Compilieren mit einer Taste oder einem Mausklick gestartet werden kann. Ein Beispiel für eine speziell für den 8051-Prozessor geeignete Entwicklungsumgebung findet man in **MC-51**. Es enthält einen Texteditor und integriert einen Assembler (ASEM-51) und als Pascal-Compiler Turbo51. Außerdem enthält es noch einen Simulator mit Quelltext-Debugger.

Beim Starten des Compilers ist es immer zu empfehlen, die Befehlszeilen-Option -M zu benutzen, um veränderte Quelltexte auch von Units, die im Projekt benutzt werden, automatisch neu zu übersetzen. Eine vollständige Neukompilierung aller verwendeten Module wird mit der Option -B veranlasst.

Turbo51 hält sich weitgehend an die Syntax von Borland Turbo Pascal 7 einschließlich der objekt-orientierten Programmierung (OOP), benutzt aber einige zusätzliche Direktiven und Konstrukte, um die speziellen Eigenschaften der 8051-Familie (MCS-51) zu unterstützen.

Reservierte Worte:

```
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
```

Direktiven:

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

Dieses Handbuch ist eine deutsche Übersetzung der Internet-Seite von Turbo-51:
http://turbo51.com/documentation

2 Allgemeines

2.1 Befehlszeilen-Syntax

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

Option	***Beschreibung***
-A	Erzeuge eine Assembler-Datei
-B	Alle Module des Projektes neu erzeugen (Build)
-C	Anzeige der Spaltenposition eines Fehlers
-D*symbols*	Bedingungen definieren
-E*path*	Ausgabeverzeichnis für BIN/HEX/U51/OMF
-F*hex address*	Quellzeile bei einer Adresse finden
-G	Map-Datei erzeugen
-H	Hex-Intel-Datei erzeugen
-I*path*	Verzeichnis für Include-Dateien
-J*path*	Verzeichnis für Object-Dateien
-LA	Verwende die Bibliothek Turbo51A.l51 (compiliert mit $A+ für `ACALL/AJMP`-Befehle)
-M	Erzeuge nur geänderte Units
-MG*memory type*	Standard-Speichertyp für globale Variablen (*memory type* = D, I oder X)
-ML*memory type*	Standard-Speichertyp für lokale Variablen (*memory type* = D, I oder X)
-MP*memory type*	Standard-Speichertyp für Parameter-Variablen (*memory type* = D, I oder X)
-MT*memory type*	Standard-Speichertyp für temporäre Variablen (*memory type* = D, I oder X)
-O	Erzeuge OMF-51-Datei
-OX	Erzeuge erweiterte OMF-51-Datei (für quelltext-basierte Fehlersuche)

-Q	Compiliere ohne Meldungen (Quiet)
-S	Syntaxprüfung
-T*path*	Verzeichnis für L51/CFG
-U*path*	Verzeichnis für Units
-$*directive*	Compiler-Befehle in der Befehlszeile

Es ist immer zu empfehlen, beim Compilieren die Option -M zu verwenden, damit nur die Units neu übersetzt werden, die verändert wurden. Das Compilieren wird dadurch beschleunigt. Für den Fall, dass alle Units neu übersetzt werden sollen, verwendet man die Option -B. Beim Compilieren wird für jede Unit eine Datei vom Type 'UnitName.u51' erzeugt, die beim nachfolgenden Compilieren des Hauptprogramms eingebunden wird. Ohne eine der Optionen -M oder -B würden jedes Mal beim Compilieren alle Units neu übersetzt, ohne dass die übersetzte Unit-Datei (U51) gespeichert würde.

Compiler-Schalter in der Befehlszeile:

(Standardwerte siehe 2.2)

Compiler-Befehl	***Beschreibung (wenn gesetzt)***
-$A-	Erzeuge absolute Befehle (`ACALL/AJMP`)
-$B-	Vollständige Auswertung von boolschen Ausdrücken
-$C+	Quelltextzeilen im Assembler-Code anzeigen
-$I+	IDATA-Variablen dürfen unter $80 beginnen (wie bei indirekt adressierten DATA-Variablen)
-$O+	Optimierung
-$P-	Offene String-Parameter
-$R-	Ablaufinvariante Prozeduren (Reentrant)
-$T-	Typisierte Zeiger
-$U-	Eindeutige lokale Variablennamen
-$V+	Strenge Var-String-Prüfung
-$X+	Erweiterte Syntax

2.2 Compiler-Schalter und -Direktiven

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

(Standardwerte sind jeweils angegeben)

Compiler-Schalter	***Beschreibung (wenn gesetzt)***
$A-	Erzeuge absolute Befehle (`ACALL/AJMP`)
$AbsoluteInstructions *Off*	dto.
$B-	Vollständige Auswertung von boolschen Ausdrücken
$C+	Quelltextzeilen im Assembler-Code anzeigen
$DefaultFile *Off*	Der Dateisystem-Variablen *CurrentIO* werden im aktuellen Programme I/O-Prozeduren zugewiesen
$I+	IDATA-Variablen dürfen unter $80 beginnen (wie bei indirekt adressierten DATA-Variablen)

$InlineCode *On*	Vom Compiler wird normaler Inline-Code erzeugt.
$NoReturn	Verhindert die Erzeugung eines `RET`-Befehls innerhalb einer Assembler-Prozedur
$O+	Optimierung
$P-	Offene String-Parameter
$R-	Ablaufinvariante Prozeduren (Reentrant)
$T-	Typisierte Zeiger
$U-	Eindeutige lokale Variablennamen
$V+	Strenge var-String-Prüfung
$X+	Erweiterte Syntax

Hinweis: Zwischen dem Buchstaben und dem + oder – darf kein Zwischenraum stehen, vor *On* und *Off* muss ein Zwischenraum stehen.

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

Compiler-Direktiven: `{$<directive><value> }`

Compiler-Direktive	***Beschreibung***
$DEFINE	Definiere ein Symbol für die bedingte Kompilierung
$ELSE	Bedingte Kompilierung mit `IFDEF` und `IFNDEF`
$ENDIF	Ende der bedingten Kompilierung
$IFDEF *symbol*	Bedingte Kompilierung, wenn *symbol* definiert ist (s.o.)
$IFNDEF *symbol*	Bedingte Kompilierung, wenn *symbol* nicht definiert ist (s.o.)
$IFOPT *switch*(+/-)	Bedingte Kompilierung, wenn ein Compiler-Schalter gesetzt/nicht gesetzt ist
$M	Speichergrößen (nur im Hauptprogramm), angegeben sind die Standardwerte:
CODE Start,	$0000
CODE Size,	$10000
XDATA Start,	$0000
XDATA Size,	$0000
Heap Size	$0000
$IDATA	IDATA ist verfügbar
$XDATA	XDATA ist verfügbar (nur in einer Unit)
$HEAP	Ein Heap ist verfügbar (nur in einer Unit)
$MG *memory type*	Setze den Standardspeichertyp für globale Variablen (*memory type* = *DATA, IDATA* oder *XDATA*)
$ML *memory type*	Setze den Standardspeichertyp für lokale Variablen (*memory type* = *DATA, IDATA* oder *XDATA*)
$MP *memory type*	Setze den Standardspeichertyp für Parameter-Variablen (*memory type* = *DATA, IDATA* oder *XDATA*)
$MT *memory type*	Setze den Standardspeichertyp für temporäre Variablen (*memory type* = *DATA, IDATA* oder *XDATA*)

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

2.3 Speicherorganisation

CODE-Speicher

Per Standardvorgabe beträgt die maximale Codespeichergröße $10000 Bytes (64 KB). Dies kann im Hauptprogramm durch die Direktive **$M** geändert werden.

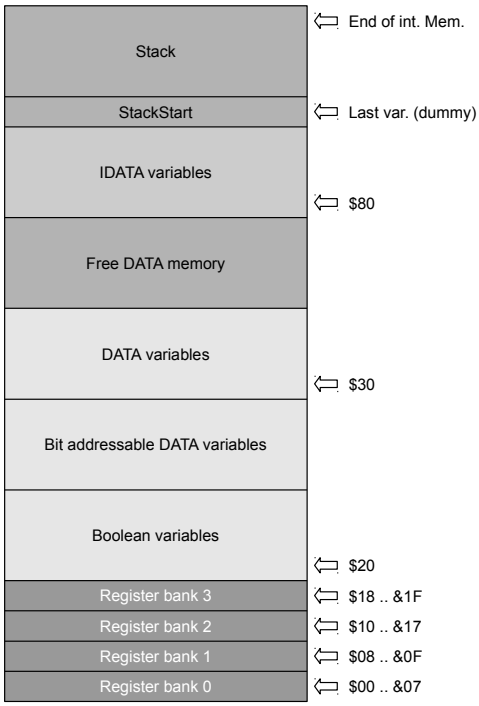

Abb. 1: Interner Speicher

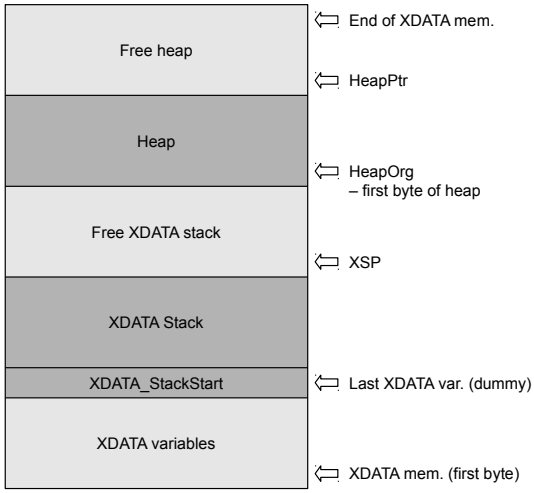

Abb. 2: Externer Speicher

IDATA-Speicher

Per Standardvorgabe ist kein IDATA-Speicher vorhanden. Dies kann im Hauptprogramm oder indirekt benutzten Units durch die Direktive **$IDATA** geändert werden. Üblicherweise verwendet das Hauptprogramm eine Unit, in der die Merkmale des Mikrocontrollers festgelegt werden. Dazu gehört auch die Direktive **$IDATA**, wenn ein IDATA-Speicher zur Verfügung steht.

XDATA-Speicher

Per Standardvorgabe ist kein XDATA-Speicher vorhanden. Dies kann im Hauptprogramm oder einer benutzten Unit durch die Direktive **$M** geändert werden.

Speicher-Organisation von DATA / IDATA

Wenn der Compiler-Schalter **$I** gesetzt ist, folgen die IDATA-Variablen unmittelbar den DATA-Variablen. Wenn es keinen IDATA-Speicher oder keine IDATA-Variablen gibt, folgt der Stack unmittelbar den DATA-Variablen. (Abb. 1).

Speicher-Organisation von XDATA

Die Anfangsadresse und die Größe des XDATA-Speichers und des Heaps können mit der Compiler Direktive **{$M***CODE Size, XDATA Start, XDATA Size, Heap Size***}** gesetzt werden (Abb. 2).

2.4 System-Unit

In der System-Unit ist die Turbo51-Laufzeit-Bibliothek enthalten, und es sind die Special-Function-Registers (SFR), Bit- und Interruptadressen definiert, die in allen auf dem 8051-Kern basierten Mikrocontrollern verwendet werden. Sie wird normalerweise aus der Datei *Turbo51.L51* geladen. Es handelt sich dabei um eine Sammlung von compilierten Units.

Daneben gibt es noch eine *Turbo51A.L51*-Bibliothek, die die System-Unit mit dem **$A+** compilierten Schalter enthält und keine LCALL/LJMP-Befehle enthält (um sie zu benutzen, muss die Befehlszeilen-option **-LA** verwendet werden). Hinweis: Bis der Compiler ein stabileres Stadium erreicht hat, können sich einige Deklaration in dieser Unit noch ändern.

Zusätzliche Definitionen für andere Mikrocontroller der 8051-Familie befinden sich in verschiedenen zusätzlichen Units mit den Namen wie *Sys_xxxx.pas* (z.B. *Sys_80C592.pas*, *Sys_89S8253.pas*).

Alle Definitionen der System-Unit sind im **Anhang A** aufgeführt.

2.5 Dateien

Für die Abwicklung des Datentransfers von und zu angeschlossenen Geräten verwendet Turbo51 Datei-Variablen. Die eigentlichen IO-Prozeduren müssen dabei vom Benutzer bereitgestellt werden. Es gibt drei verschiedenene Arten von Datei-Variablen: *untypisiert*, *typisiert* (Datei eines bestimmten Datentyps) und *text-basiert* (ASCII-Text mit Zeilenvorschub (Line-Feed = #10) als Begrenzer). Sie können mit folgenden Prozeduren benutzt werden:

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

Im **Anhang B** befindet sich ein einfaches Beispiel für einen Rechner, der mit Datei-Variablen für die Ein- und Ausgabe arbeitet.

2.6 Objekte

Objekte sind Daten-Strukturen, die Pascal-Records und Prozeduraufrufe kombinieren, d.h. Daten und Code. Um sie in Turbo51 zu verwenden, benötigt man XDATA-Speicher. Die Syntax ist ähnlich, wie die in Borland Turbo Pascal 7 verwendete. Turbo51 unterstützt folgende Merkmale:

- Vererbung (Inheritance)
- Statische und dynamische Objekte
- Private Felder
- Konstruktoren und Destruktoren
- Statische, virtuelle und dynamische Methoden

Siehe **Anhang C** für ein Beispiel.

3 Deklarationen

3.1 Konstanten

Turbo51-Konstanten dürfen jeden ordinalen Typ darstellen. Typisierte Konstanten werden im CODE-Speicher abgelegt (im „little-endian Format“, d.h. das erste Byte enthält den niederwertigsten Teil) und

können nicht verändert werden. Es ist nicht möglich boolesche typisierte Konstanten zu verwenden, da Variablen vom Type *boolean* im bit-adressierbaren Bereich des DATA-Speichers abgelegt werden (ist in allen 8051-artigen Prozessoren vorhanden). Möglich ist dagegen eine typisierte *ByteBool*- oder andere ähnlich Konstante.

Einige Beispiele befinden sich im **Anhang D**.

3.2 Typen

In Turbo51 stehen folgende System-Typen zur Verfügung: *Byte* (vorzeichenloses 8-bit), *Word* (vorzeichenloses 16-bit), *ShortInt* (vorzeichenbehaftetes 8-bit), *Integer* (vorzeichenbehaftetes 16-bit), *LongInt* (vorzeichenbehaftetes 32-bit), *Real* (uses 4 bytes), *String*, *Boolean*, *ByteBool*, *WordBool*, *LongBool* and *Char*. Entsprechend der Pascal-Syntax können darauf aufbauend beliebige andere Typen konstruiert werden.Es gibt in Turbo51 drei Zeigertypen: *ShortPtr* (zeigt auf IDATA), *Pointer* (zeigt auf XDATA) and *CodePointer* (zeigt auf CODE). Entsprechend gibt es *ShortPChar, PChar* and *CodePChar*. Zeiger auf ordinale Typen können mit einer Speichertyp-Direktive DATA, IDATA oder XDATA den Standard-Speicherort für die zugehörigen Variablen umdefinieren.

Im **Anhang E** befinden sich einige Beispiele.

3.3 Variablen

Variable können in Turbo51 mit einer Speichertyp-Direktive DATA, IDATA oder XDATA versehen werden, die den Standard-Speicherort umdefiniert (ein IDATA-Speicher mit einer Adresse über $80 ist nicht bei allen Mikrocontrollern der 8051-Familie vorhanden, einige haben zusätzlich einen internen XDATA-Speicher). Boolesche Variablen werden immer im bit-adressierbaren Bereich des DATA-Speichers abgelegt. Dieser ist bei allen 8051-artigen Prozessoren vorhanden). Die *Volatile*-Direktive definiert nicht zu optimierende Variablen, weil sie z.B. in einer Interrupt-Prozedur verändert werden. Die *Absolute*-Direktive definiert eine Variable die einer anderen überlagert (z.B. auch *AbsVar absolute RecordVariable.Field*) oder einer festen Adresse zugeordnet ist. Boolesche Variablen können nicht als Referenz übergeben werden (der 8051 hat keinen Befehl um ein Bit per indirekter Adresse anzusprechen) und auch nicht als Parameter für ablaufinvariante (reentrant) Prozeduren. In diesen Fällen muss man den Typ *ByteBool* verwenden, der 1 Byte im Datenspeicher belegt. Die Direktive *BitAddressable* definiert eine Variable vom Typ *Byte* oder *ShortInt*, die sich im DATA-Speicher zwischen den Adressen $20 und $2F befindet und damit bitadressierbar wird. Man kann somit über BitAddressableVar.n (n = 0,..,7) auf einzelne Bits dieser Variablen zugreifen. Variablen, die mehrere Bytes belegen, werden immer im „little-endian Format“ abgespeichert (d.h. das erste Byte enthält den niederwertigsten Teil).

Einige Beispiele dazu befinden sich im **Anhang F**.

Es ist auch möglich, in einer Unit Variablen mit der Direktive *absolute Forward* zu definieren, was bedeutet, dass die Adresse innerhalb der Unit unbekannt ist:

```
Unit I2C;

Interface

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

Die Deklaration der zugehörigen Adressen erfolgt in dem Hauptprogramm, das diese Unit verwendet:

```
Program Test;
```

```
Uses I2C;

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

4 Prozeduren

4.1 System-Prozeduren

Assign

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

Die Prozedur *Assign* weist einer Datei-Variablen *F* eine Lesefunktion und eine Schreibprozedur zu. Im Aufruf dürfen sowohl *ReadFunction* als auch *WriteProc* weggelassen werden. Die Lesefunktion muss eine nicht ablaufinvariante Funktion ohne Parameter sein und **muss** die Register `R2, R3, R4, R5, R8, R9` sichern. Sie liefert einen Wert vom Typ *Char* oder *Byte* zurück, der im Register A zurück gegeben wird, wie es standardmäßig bei allen Turbo51-Pascal-Funktionen der Fall ist.
Die Schreibprozedur muss eine nicht ablaufinvariante Prozedur ohne Parameter sein und **muss** die Register `R2, R3, R6, R7` sichern. Das zu schreibende Zeichen wird der Prozedur im Register A übergeben. Wenn diese Prozedur in Pascal (nicht im Assembler) geschrieben ist, muss das Zeichen zunächst in einer lokalen Variablen zwischengespeichert werden (*asm*-Anweisung am Anfang der Prozedur).

BlockRead

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

Die Prozedur *BlockRead* liest *Count* Bytes aus der Datei-Variablen *F* in den Speicherbereich, der durch die Variable *Buffer* definiert ist. Die Datei wird über die oben beschriebene *ReadFunction* eingelesen, die der Datei-Variablen *F* zugewiesen wurde.

BlockWrite

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

Die Prozedur *BlockWrite* schreibt *Count* Bytes aus dem durch die Variable *Buffer* beschriebenen Speicherbereich in die Datei-Variable *F.* Die Bytes werden von der *WriteProcedure* geschrieben, die der Datei-Variablen *F* zugewiesen wurde (siehe unter *Assign*).

Break

```
Procedure Break;
```

Break springt zu der Anweisung, die dem Ende der laufenden Schleife folgt. Der Code zwischen der *Break*-Anweisung und dem Ende der Schleife wird übersprungen, die Schliefe wird beendet. Diese Prozedur kann mit *For-*, *Repeat-* und *While*-Anweisungen verwendet werden.

Change

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

Change ändert das Vorhandensein von *Element* in der Menge *S.* Wenn das Element enthalten war, wird es entfernt und umgekehrt.

Continue

```
Procedure Continue;
```

Continue springt an das Ende der aktuellen Schleife. Der Code zwischen dem *Continue- Anweisung* und dem Ende der Schleife wird übersprungen, die Schleife wird fortgesetzt. Diese Prozedur kann mit *For-*, *Repeat-* und *While-*Anweisungen verwendet werden.

Dec

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

Dec erniederigt den Wert von *X* um *Decrement*. Wenn *Decrement* nicht angegeben ist, wird 1 als Vorgabewert verwendet.

Delete

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

Delete löscht *Count* Zeichen aus dem String *S* beginnend an der Position *Index*. Alle Zeichen hinter den gelöschten werden um *Count* Positionen nach links verschoben. Die Länge des Strings wird angepasst.

Dispose

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

Die erste Form von *Dispose* gibt den mit *New* zugewiesenen Speicherplatz auf dem Heap wieder frei. Die zweite Form von *Dispose* erwartet als ersten Parameter einen Zeiger auf ein Objekt und als zweiten den Namen der Destruktor-Methode dieses Objekts. Der Destruktor wird aufgerufen und anschließend der zugewiesene Speicher freigegeben.

Exclude

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

Exclude schließt *Element* von der Menge *S* aus.

Exit

```
Procedure Exit;
```

Exit verlässt die aktuelle Prozedur oder Funktion und kehrt zur aufrufenden Routine zurück.

ExitBlock

```
Procedure ExitBlock;
```

ExitBlock verlässt den aktuellen „*begin*-end“-Block und springt an die erste Anweisung außerhalb dieses Blocks.

Fail

```
Procedure Fail;
```

Fail verlässt einen Konstruktor mit dem Wert *nil*.

FillChar

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

Fillchar füllt den Speicher beginnend bei *Mem* mit *Count* Zeichen des Wertes *Value*.

FreeMem

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

FreeMem gibt den für den Zeiger *Ptr* in der Größe *Count* (in Bytes) belegten Speicherplatz auf dem Heap frei. *Ptr* sollte auf den Speicherbereich zeigen, der mit der Prozedure *GetMem* einer dynamischen Variablen zugewiesen wurde.

GetMem

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

GetMem belegt *Size* Bytes Speicher auf dem Heap und liefert als Variable *Ptr* einen Zeiger auf diesen Speicherbereich zurück. Wenn kein Speicher mehr verfügbar ist, wird *nil* zurückgegeben.

Halt

```
Procedure Halt;
```

Halt generiert Code für eine Endlos-Schleife (d.h. einen Sprung auf sich selbst).

Inc

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

Inc erhöht den Wert von *X* um *Decrement*. Wenn *Decrement* nicht angegeben ist, wird 1 als Vorgabewert verwendet.

Include

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

Include schließt *Element* inn die Menge *S* ein.

Insert

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

Insert fügt den String *Source* in den String *DestStr* an der Position *Index* ein. Dabei werden alle Zeichen hinter *Index* nach rechts geschoben. Der sich ergebende String wird, falls erforderlich, bei 255 Zeichen abgeschnitten.

Mark

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

Mark kopiert den aktuellen Heap-Zeiger nach *Ptr*.

Move

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

Move schiebt *Count* Bytes von *Source* nach *Dest*.

New

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

New belegt Speicherplatz für den typisierten Zeiger *Ptr*. Die Adresse wird in *Ptr* übergeben. Wenn *Ptr* auf ein Objekt zeigt, ist es möglich, den Namen des zugehörigen Konstruktors anzugeben.

Randomize

```
Procedure Randomize;
```

Randomize initialisieret den Zufallszahlen-Generator von Turbo51. Dabei wird in der Systemvariablen *RandSeed* ein aus dem Systemtakt abgeleiteter Wert gespeichert.

Read

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

Read liest einen oder mehrere Werte aus der Datei-Variablen *F* und weist sie den Variablen *V1*, *V2*, etc. zu. Wenn keine Datei-Variable *F* angegeben ist, wird von der Standard-Eingabe gelesenen. Wenn *F* für eine typisierte Datei steht, müssen alle Variablen dem Typ entsprechen, mit dem *F* deklariert wurde.

Readln

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

Readln liest einen oder mehrere Werte aus der Textdatei-Variablen *F* und weist sie den Variablen *V1*, *V2*, etc. zu. Danach wird in die nächste Zeile (definierte durch einen Zeilenvorschub (Linefeed = `#10`) gesprungen. Wenn keine Datei-Variable *F* angegeben ist, wird von der Standard-Eingabe gelesenen.

Release

```
Procedure Release (Ptr: Pointer);
```

Release setzt die Heapgrenze auf den durch *Ptr* angegebenen Wert zurück. Speicher oberhalb der durch *Ptr* angegeben Adresse wird als leer gekennzeichnet.

Str

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

Str liefert einen String zurück, der den Wert von *X* beschreibt. *X* darf ein beliebiger numerischer Typ sein. Die optionalen Angaben *NumPlaces* und *Decimals* legen die Formatierung des Strings fest.

Val

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

Val konvertiert den durch einen String *Str* beschriebenen Wert in eine Zahl und speichert sie unter *V*. Die Variable *V* kann vom Typ *Longint* oder Real sein. Wenn die Konvertierung fehlschlägt, enthält der Parameter *ErrorCode* den Index des Zeichens, das den Fehler verursachte, sonst ist er 0. Der String *Str* darf keine Leerzeichen enthalten.

Write

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

Write schreibt die Werte der Variablen *V1*, *V2* etc. in die Datei-Variable *F*. *F* kann eine typisierte Datei oder eine Textdatei sein. Wenn es sich um eine typisierte Datei handelt, müssen die Variablen *V1*, *V2,..* zu der Typdeklaration der Datei passen. Untypisierte Dateien sind nicht erlaubt. Wenn der Parameter *F* weggelassen wird, wird in die Standardausgabe geschrieben (dies ist die System-Variable *Output*, ein Synonym für die Textdateivariable *SystemIO*). Wenn es sich bei *F* um den Typ *Text* handelt, werden bei den übergebenen Variablen alle erforderlichen Umwandlungen für eine Textausgabe vorgenommen.

Dabei sind alle numerischen Formate möglich. Strings und *PChar*-Typen werden exakt so ausgegeben, wie sie im Speicher vorliegen. Das Format von numerischen Ausgaben kann durch weitere Parameter beeinflusst werden: *OutputVariable: NumChars [: Decimals]*. Die Ausgabe erfolgt dann mit mindestens *NumChars* Zeichen, *Decimals* gibt die Anzahl der Stellen hinter dem Komma an. Wenn die Zahl *NumChars* zu klein für die Darstellung des Wertes ist, wird dieser automatisch angepasst, wenn weniger Stellen benötigt werden, wird auf der linken Seite mit einer passenden Anzahl von Leerzeichen aufgefüllt, so dass die Ausgabe rechtsbündig erscheint. Wird keine Formatierung angegeben, erfolgt die Ausgabe der in der jeweils erforderlichen Länge, bei positiven Zahlen ohne weiteren Vorsatz, bei negativen mit einem Minus. Fließkommazahlen (*Real*) werden in der wissenschaftlichen Notierung ausgegeben.

Writeln

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

Writeln arbeitet genau so wie *Write* für Text-Dateien. Zusätzlich wird ein Wagenrücklauf/Zeilenvorschub (Carriage Return - LineFeed = `#13#10`) ausgegeben. Wenn der Parameter *F* weggelassen wird, wird in die Standardausgabe geschrieben (s.o.), wenn keine Variablen angegeben sind, wird nur ein Wagenrücklauf/Zeilenvorschub geschrieben.

4.2 System-Funktionen

Abs

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

Abs gibt den Absolutwert der Variablen X zurück. Das Ergebnis ist vom selben Typ wie das Argument (*Integer* oder *Real*).

Addr

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

Addr liefert einen Zeiger auf des jeweilige Argument, das von einem beliebigen Typ einschließlich Prozedur und Funktion sein darf. Wenn das Argument für eine Variable im DATA-Speicher steht, ist das Ergebnis vom Typ *ShortPtr*, wenn es im XDATA-Speicher steht, vom Typ *Pointer,* und wenn es im CODE-Speicher steht (typisierte Konstante, Prozedur, Funktion oder statische Methode), ist das Ergebnis vom Typ *CodePointer*. Das Ergebnis ist zuweisungskompatibel zu allen Zeigertypen. Im Gegensatz dazu liefert der Operator @ einen typ-spezifischen Zeiger.

ArcTan

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

Arctan gibt den Arcustangens von *X* zurück. Als Winkeleinheit wird das Rad benutzt.

Assigned

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

Assigned liefert *True,* wnn *P* nicht *nil* ist und *False* im anderen Fall. *P* darf ein belibiger Zeiger oder eine prozedurale Variable sein.

Bcd

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

Bcd erzeugt das binär kodierte dezimale Äquivalent von *D*.

Chr

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

Chr ist das ASCII-Zeichen, das Wert *X* hat.

Concat

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

Concat verbindet die Strings *S1*, *S2* etc. zu einem langen String. Das Ergebnis wird auf eine Länge von 255 Bytes abgeschnitten. Das gleiche Ergebnis lässt sich mit dem +-Operator erzielen. Die Funktion *Concat* benötigt XDATA-Speicher.

Copy

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

Copy liefert einen Teilstring zurück, der aus *Count* Zeichen beginnend bei der Position *Index* besteht. Ist der Wert von *Count* größer als die Länge des Strings, wird das Ergebnis entsprechend abgeschnitten. Wenn *Index* größer ist als die Länge des Strings, wird ein leerer String zurückgegeben. Die Funktion *Copy* benötigt XDATA-Speicher.

Cos

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

Cos berechnet des Kosinus des durch *X* in *Rad* angegebenen Winkels.

Exp

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

Exp berechnet *e* hoch *X*.

Frac

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

Frac liefert den nicht ganzzahligen Teil der Fließkommazahl *X* zurück.

Hi

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

Hi liefert das höherwertige Byte von *X*.

High

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

Das Ergebnis der Funktion *High* hängt von dem jeweiligen Argument ab:

1. Wenn das Argument ein Ordinaltyp ist, liefert *High* den höchsten Wert innerhalb des Bereichs des Ordinaltypen zurück.
2. Wenn das Argument ein Array-Typ ist, liefert *High* den höchsten möglichen Wert seines Index zurück.
3. Wenn das Argument ein offenes Array in einer Prozedur oder Funktion ist, liefert *High* den höchsten Index des bei Index 0 beginnenden Arrays zurück. Der Rückgabewert ist vom gleichen Typ wie der Typ wie der des Array-Indexes.

Int

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

Int liefert den ganzzahligen Anteil der Fließkommazahl *X*. Das Ergebnis ist vomTyp *Real*, d.h. auch eine Fließkommazahl.

Length

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

Length liefert die Länge des Strings *S* zurück. Bei einem leeren String *S* wird 0 zurückgegeben. Die Stringlänge wird auch in *S [0]* gespeichert.

Ln

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

Ln berechnet den natürlichen Logarithmus der positiven Fließkommazahl *X*.

Lo

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

Lo liefert das niederwertige Byte von *X*.

Low

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

Das Ergebnis der Funktion *Low* hängt von dem jeweiligen Argument ab:

1. Wenn das Argument ein Ordinaltyp ist, liefert *Low* den kleinsten Wert innerhalb des Bereichs des Ordinaltypen zurück.
2. Wenn das Argument ein Array-Typ ist, liefert *Low* den kleinsten möglichen Wert seines Index zurück.
3. Wenn das Argument ein offenes Array in einer Prozedur oder Funktion ist, liefert *Low* den kleinsten Index des Arrays (immer 0) zurück. Der Rückgabewert ist vom gleichen Typ wie der Typ wie der des Array-Indexes.

MaxAvail

```
Function MaxAvail: Word;
```

MaxAvail liefert die Anzahl der Bytes des größten verfügbaren freien Speicherblocks auf dem Heap zurück.

MemAvail

```
Function MemAvail: Word;
```

MemAvail liefert die Anzahl aller Bytes zurück, die auf dem Heap nicht belegt sind.

New

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

New reserviert für den Zeigertyp *PType* Speicher auf dem Heap und liefert einen Zeiger mit der Adresse darauf zurück. Wenn *PType* ein Zeiger auf ein Objekt ist, kann der Name der Konstruktor-Methode dieses Objekts angegeben werden.

Odd

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

Odd liefert *True,* wenn *X* ungerade ist, sonst *False*.

Ofs

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

Ofs liefert den Offset von *Field* innerhab des Records *TRecord*.

Ord

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

Ord liefert den Oridinalwert der ordinalen Variable *X* zurück.

Pi

```
Function Pi: Real;
```

Pi liefert den Wert der Zahl π (3.1415926535897932385).

Pos

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

Pos liefert die Position von *Substr* innerhalb des Strings *Str*, sofern *Substr* in *Str* enthalten ist. *Andernfalls wird* 0 zurückgegeben. Die Suche beachtet die Groß- und Kleinschreibung.

Pred

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

Pred liefert den Vorgänger der ordinalen Variablen *X* zurück.

Random

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

Random berechnet eine Zufallszahl $0 \leq$ Ergebnis $< L$. Wenn das Argument *L* weggelassen wird, ist das Ergebnis eine Fließkommazahl $0 \leq$ Ergebnis < 1.

Round

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

Round rundet *X* auf den nächsten ganzzahligen Wert. Liegt der Wert *X* exakt zwischen zwei ganzen Zahlen wird in Richtung des größeren Wertes gerundet.

Sin

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

Sin berechnet den Sinus zu dem in *Rad* angegeben Winkel *X*.

SizeOf

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

SizeOf liefert die Größe einer Variablen in Bytes zurück.

Sqr

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

Sqr berechnet das Quadrat des Arguments *X*.

Sqrt

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

Sqrt berechnet die Quadratwurzel des positiven Arguments *X*.

Succ

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

Succ liefert den Nachfolger der ordinalen Variablen *X* zurück.

Swap

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

Swap vertauscht die beiden Bytes (8 Bit) von *X*.

SwapWord

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

SwapWord vertauscht die beiden Wortanteile (16 Bit) von *X*.

Trunc

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

Trunc liefert den ganzzahligen Anteil von *X* zurück.

TypeOf

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

TypeOf liefert die Adresse der VMT (virtual methode table) von *TObjectType*.

UpCase

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

UpCase liefert den zu *C* gehörenden Großbuchstaben (ASCII-Code 0..127).

4.3 Assembler-Prozeduren

Im **Anhang G** findet man einige Beispiele für vollständig in Assembler geschriebene Prozeduren. Turbo51 fügt am Ende jeweils nur einen *RET*–Befehl (bzw. RETI bei Interrupt-Prozeduren) hinzu. Wenn sich Ende einer Prozedur ein RET-Befehl befindet, der nie erreicht wird, wird dieser vom Turbo51-Compiler automatisch entfernt. Soll aus irgendeinem Grund dieser Befehl nicht entfernt werden, kann man die Compiler-Direktive **$NoReturn** innerhalb der Assembler-Prozedur verwenden. Parameter werden per Wert (Turbo51 erzeugt automatisch statische Variablen, um die Werte zu speichern), per Referenz (Turbo51 erzeugt automatisch statische Variablen, um die Zeiger zu speichern) oder über die Register übergeben. Die Prozedur-Parameter werden als lokale Variable mit dem Namen *Procedure.Parameter* angesprochen (siehe auch 5.1).

4.4 Inline-Prozeduren

Prozeduren (und Funktionen) die mit der Inline-Direktive deklariert werden, werden an die jeweiligen Plätze im Programm kopiert, wo sie aufgerufen werden. Dadurch gibt es an dieser Stelle keinen Unterroutinenaufruf. Das Ergebnis ist eine schnellere Ausführung, insbesondere wenn die betroffene Prozedur oder Funktion sehr häufig benötigt wird, aber auch einen längeren Programmcode. Mit der **$InlineCode**-Direktive kann dieses Verhalten außer Kraft gesetzt werden. Wird sie auf *Off* gestellt (Standardwert ist *On*), erzeugt der Compiler normale Aufrufe für die Prozeduren.

Ein Beispiel dazu befindet sich im **Anhang H**.

4.5 Absolute Prozeduren

Mit der *absolute*-Direktive kann man erreichen, dass einer Prozedur eine feste Adresse zugewiesen wird. Auf diese Weise kann man auch bei einer festen Adresse eine Anzahl von Bytes im CODE-Segment freihalten. Normalerweise besteht keine Veranlassung, Prozeduren feste Adressen zuzuweisen, der Linker regelt dies automatisch.

Im **Anhang I** befinden sich einige Beispiele.

4.6 Interrupts

Interrupts sind Prozeduren, denen über die *Interrupt*-Direktive feste Interrupt-Adressen zugewiesen werden. So ist z.B. *Timer0* eine in der System-Unit definierte Konstante für die über die Hardware direkt angesprungene dazugehörige Interrupt-Routine. Zusätzlich kann in jeder solchen Prozedur mit der Direktive *Using* die Registerbank (0 bis 3) definiert werden, mit der in der Interrupt-Routine gearbeitet werden soll. Benutzt eine Interruptroutine keine Register, wird das mit *UsingAny* angegeben.

Warnung: Man sollte sich vergewissern, dass alle Variablen, die durch eine Interrupt-Prozedur verändert werden könne, mit der Direktive *Volatile* gekennzeichnet werden. Dies sagt dem Compiler, dass ihr Wert außerhalb des normalen Programmflusses verändert werden kann und daher viele Optimierungen nicht auf sie angewandt werden dürfen. Man sollte es vermeiden, in einer Interrupt-Routine zeitaufwändige Operationen durchzuführen, wie z.B. Fließkomma-Operationen, Datei-Ein- und -Ausgabe, String-Verarbeitung oder große Speicherumlagerungen.

Ein Beispiel für eine Interrupt-Deklaration befindet sich im **Anhang J**.

5 Assembler

5.1 Assembler-Anweisungen

Eine Turbo51-Assembler-Anweisung sieht sehr ähnlich aus wie im 8051-Assembler. Man kann alle Befehle des Standard-Befehlssatzes verwenden. Sprungadressen (*Labels*), mit einem „@“ beginnen, müssen nicht deklariert werden. Es müssen keine Register gesichert werden, ihr Inhalt ist ungewiss. Die Byte-Variablen *AR0* to *AR7* stehen für die Register `R0` to `R7` mit den zugehörigen Adressen zwischen $00 und $1F, abhängig von der jeweils aktiven Registerbank. Um auf einen Bezeichner zuzugreifen, dessen Name identisch mit einem Register ist, muss ihm ein „&“ vorangestellt werden (z.B. verwendet man *&R0,* um auf eine Variable mit dem Namen *R0* und nicht auf das Register `R0` zuzugreifen). Prozedur-Parameter können wie lokale Variablen behandelt werden (*Procedure.Parameter*).

Siehe dazu auch den Abschnitt über Assembler-Prozeduren (4.3). Ein Beispiel dazu befindet sich im **Anhang K**.

Weitere Hinweise:

`DB`	reserviert Bytes.
`DW`	reserviert Worte (jeweils 2 Bytes)
`DD`	reservierte doppelte Worte (LongInt = 4 Bytes)
`OR`	wird für das logische Oder verwendet
`AND`	wird für das logische Und verwendet
`XOR`	wird für das logische Exklusiv-Oder verwendet
`NOT`	wird für die logische Nicht-Operation verwendet operation.
`MOD`	ist der Operator für eine Modulo-Division
`SHR`	ist der Operator für das bitweise Schieben nach rechts
`SHL`	ist der Operator für das bitweise Schieben nach links
`LOW (`*`Word`*`)`	ermittelt das niederwertige Byte eines Wortes
`HIGH (`*`Word`*`)`	ermittelt das höherwertige Byte eines Wortes
`SWAP (`*`Word`*`)`	vertauscht die Bytes eines Wortes
Arithmetic functions	+, -, *, / sind die Operatoren für Integer-Arithmetik
`Procedure.Parameter`	Zugriff auf einen Parameter einer aufgerufenen Prozedur
`RecordType.Field`	Zugriff auf ein Feld in einem Record

`VMTADDRESS` *`TObjectType`*	Ermittelt die Adresse der Tabelle einer virtuellen Methode (VMT) des Objekts *TObjectType*.
`VMTOFFSET` *`TObjectType.VirtualMethod`*	Ermittelt den Offset von *VirtualMethod* innerhalb der VMT des Objekts *TObjectType*.
`VMTADDRESSOFFSET` *`TObjectType`*	Ermittelt den Offset der Adresse der VMT innerhalb des Objekts *TObjectType*.
`DMTADDRESS` *`TObjectType`*	Ermittelt de Adresse der Tabelle einer dynamischen Methode (DMT) des Objekts *TObjectType*.
`DMTINDEX` *`TObjectType.DynamicMethod`*	Ermittelt den Offset von *DynamicMethod* innerhalb der DMT des Objekts *TObjectType*.

In ablaufinvarianten Prozeduren und Funktionen (reentrant) sind folgende Symbole definiert:

`@LOCALS`	liefert den Offset der lokalen Variablen auf dem XDATA-Stack.
`@PARAMS`	liefert den Offset der lokalen Parameter auf dem XDATA-Stack.
`@RESULT`	liefert den Offset der Ergebnis-Variablen auf dem XDATA-Stack

5.2 Compiler-Interna

Wer Interesse an der internen Struktur von Turbo Pascal hat und sich den Quellcode von einigen populären, kommerziellen Pascal-Compilern ansehen möchte findet hier weitere Informationen.

5.2.1 Allgemeines

Datenspeicherung

Alle Variablen und typisierten Konstanten werden im „little-endian Format“, d.h. das erste Byte enthält den niederwertigsten Teil, abgespeichert.

Boolesche Variablen

Boolesche Variablen werden als Bits im bit-adressierbaren DATA-Speicher, den es bei allen 8051-artigen Mikrocontrollern gibt, abgelegt. Boolesche Variablen können nicht per Referenz an Prozeduren übergeben werden, da der 8051-Befehlssatz keine indirekte Adressierung von Bits vorsieht. Ebenso ist die Übergabe als Parameter in ablaufinvariante (reentrant) Prozeduren nicht möglich. In diesen Fällen kann stattdessen der Typ *ByteBool*, der 1 Byte belegt, verwendet werden.

Globale Variablen

Globale Variablen werden im ihnen durch die Compiler-Direktive **`$MG MemoryType`** (*`DATA`*, *`IDATA`* oder *`XDATA`*) and defaults to *`DATA`* zugeordneten Speicherbereich abgelegt. Der Vorgabewert ist *`DATA`*. Es ist möglich, diese Voreinstellung für jede einzelne Variablen-Deklaration zu überschreiben.

Lokale Variablen

Alle lokalen Variablen in normalen (non-reentrant) Prozeduren und Funktionen sind statisch. Sie werden wie die globalen Variablen gespeichert, können aber nur im jeweiligen lokalen Bereich angesprochen werden. Der Speicherbereich dafür wird durch die Compiler-Direktive **`$ML MemoryType`** (*`DATA`*, *`IDATA`* oder *`XDATA`*) zugewiesen, Vorgabewert ist *`DATA`*. Es ist möglich, diese Voreinstellung für jede einzelne Variablen-Deklaration zu überschreiben.

Parameter

Alle Übergabeparameter in normalen (non-reentrant) Prozeduren und Funktionen werden als lokale Variablen gespeichert und sind statisch. Durch die Compiler-Direktive **`$MP MemoryType`** (*`DATA`*, *`IDATA`* oder *`XDATA`*) wird ein Speicherbereich zugeordnet, Vorgabewert ist *`DATA`*. Es ist möglich, diese Voreinstellung für jede einzelnen Parameter zu überschreiben.

Verwendung der Register

Turbo51 benutzt intern zwei Registersätze: R5R4R3R2 und R9R8R7R6. R8 und R9 sind gewöhnlich DATA-Variablen und in der System-Unit deklariert. 8-Bit-Daten werden in R2 (R6) gespeichert, 16-Bit-Daten in R3R2 (R7R6), 32-Bit-Daten benutzen den kompletten Registersatz.

XDATA Stack

Wenn ein XDATA-Speicher vorhanden ist, erzeugt Turbo51 dort einen Stack. *XSP* (Ein in der System-Unit deklarierter Zeiger) weist auf die Spitze des Stacks.

Aufruf von normalen (nicht-reentrant) Prozeduren und Funktionen

Bei normalen, nicht ablaufinvarianten (non-rentrant) Prozeduren und Funktionen werden alle Übergabe-Parameter als lokale Variablen gespeichert. Die aufrufende Routine übergibt die Daten, indem sie sie in den zugehörigen lokalen Speicherbereich schreibt und dann die Prozedur aufruft. Funktion liefern einfache Ergebnisse entweder in ACC oder den ersten vier Registern (R5R4R3R2) zurück. String-Funktionen übergeben einen Zeiger auf das Ergebnis. Dieser befindet sich entweder in R0 (wenn der String sich im DATA- oder IDATA-Speicher befindet) oder in DPTR (wenn der String im CODE- oder XDATA-Speicher liegt). Z.Zt. sind dies die einzigen unterstützten Methode.

Meistens ist es nicht erforderlich, ablaufinvariante Prozeduren zu benutzen. Es erübrigt einen XDATA-Stack und vereinfacht den generierten Code erheblich. Wenn irgend möglich, sollte der Anwender es vermeiden, ablaufinvariante Prozeduren zu benutzen.

5.2.2 Ablaufinvariante (reentrant) Prozeduren

Bei ablaufinvarianten Prozeduren werden alle Übergabeparameter auf dem XDATA-Stack abgelegt. Einfache Ergebnisse von Funktionen werden im ersten Registersatz (R5R4R3R2) übergeben. Bei Funktionen, die einen String zurückliefern, muss die aufrufenden Routine ausreichend Platz im XDATA-Speicher reservieren und die zugehörige Adresse im Stack ablegen (siehe auch Abb. 3).

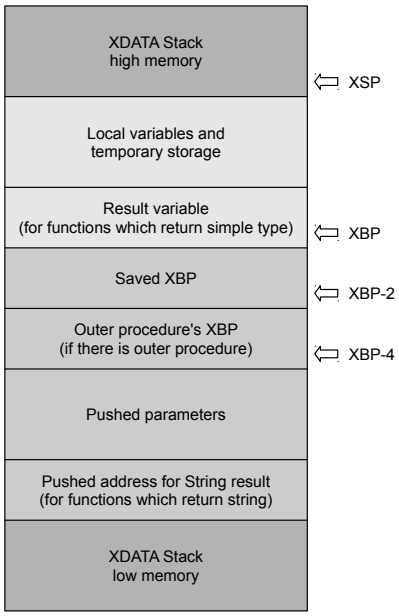

Abb. 3: Belegung des externen Speichers bei ablaufinvarianten Prozeduren

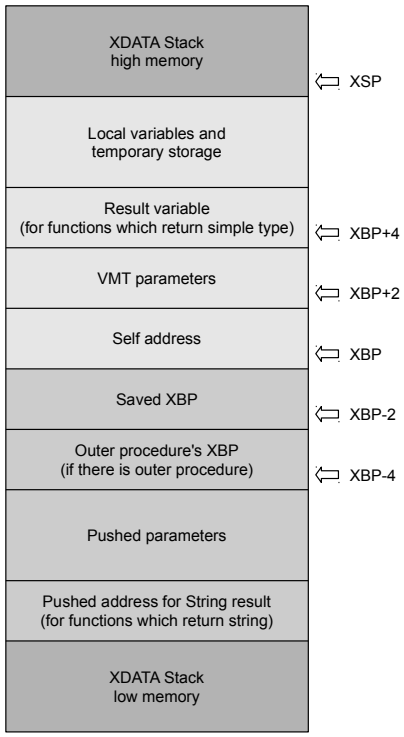

Abb. 4: Belegung des externen Speichers für Methoden

Vor dem Aufruf einer ablaufinvarianten Prozedur werden folgende Daten in den Stack geschoben (*PUSH*):

- Die Adresse für den Ergebnisstring (wenn ein solcher zurückgegeben wird)
- Alle zu übergebenden Parameter in der Reihenfolge ihrer Deklaration

- *XBP* der aufrufenden Prozedur

Die aufgerufene Prozedur muss am Anfang folgende Operationen durchführen:

- *XBP* auf dem Stack ablegen
- *XBP* neu setzen als Zeiger auf die Spitze der auf dem Stack abgelegten Paremeter
- Platz für lokale Variablen reservieren (erhöht *XSP* entsprechend)

Vor dem Verlassen der aufgerufenen Prozedur wird der gesicherte *XBP* zurückgeholt (*POP*) und entfernt alle im Stack abgelegten Parameter.

5.2.3 Methoden

Methoden sind immer ablaufinvariant. Bevor eine Methode aufgerufen wird, werden folgende Daten auf dem Stack abgelegt (*PUSH*):

- Die Adresse für den Ergebnisstring (wenn ein solcher zurückgegeben wird)
- Alle zu übergebenden Parameter in der Reihenfolge ihrer Deklaration

Bei Aufruf einer statischen Methode müssen sich in den Registern folgende Parameter befinden:

- `R3R2`: Eigene Adresse

Bei Aufruf einer virtuellen Methode müssen sich in den Registern folgende Parameter befinden:

- `DPTR`: Eigene Adresse (wird durch die System-Routine für virtuelle Methoden nach `R3R2` kopiert)
- `R2` (oder `R3R2`): Offset der VMT-Adresse
- `R0` (oder `R1R0`): Offset der Methoden-Adresse
- `R5R4`: VMT-Parameter

Beim Aufruf einer dynamischen Methode müssen sich in den Registern folgende Parameter befinden:

- `DPTR`*:* Eigene Adresse (wird durch die System-Routine für dynamische Methoden nach `R3R2` kopiert)
- `R2` (oder `R3R2`): Offset der DMT Adresse
- `R1`: Index der dynamischen Methode
- `R5R4`: VMT-Parameter

Beim Aufruf eines Konstruktors müssen sich in den Registern folgende Parameter befinden:

- `R3R2`: Eigene Adresse (Falls *nil*, wird der Konstruktor über *New* aufgerufen)
- `R5R4`: VMT-Parameter (Adresse der VMT bei normalem Aufruf, $0000 bei statischem Aufruf – keine Initialisierung der VMT-Adresse in *Self*)
- Liefert *Self* in `R3R2` zurück (Adresse des zugewiesenen Objekts)

Beim Aufruf eines Destruktors müssen sich in den Registern folgende Parameter befinden:

- `R3R2`: Eigene Adresse
- `R4`: VMT-Parameter ($00: normaler Destruktor-Aufruf, $01: Aufruf über *Dispose*)

Bei Eintritt aufgerufene Methoden:

- Sichere *XBP* auf dem Stack (Push)
- Neuer *XBP* zeigt auf die Spitze der gesicherten Parameter
- Sichere den *Self-P*arameter, der in `R3R2` übergeben wurde, auf dem Stack
- Sichere den VMT-Parameter, der in `R5R4` übergeben wurde, auf dem Stack
- Reserviere Platz für lokale Variablen (*XSP* wird entsprechend erhöht)

Beim Verlassen holt die aufgerufene Methode den gesicherten *XBP* zurück und entfernt alle anderen auf dem XDATA-Stack gesicherten Parameter. Die Anordnung im Stack während eines Aufrufs einer Methode ist in Abb. 4 dargestellt.

Anhang 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;
```

Anhang B – einfaches Beispiel für die Ein- und Ausgabe von Daten:

Das Beispiel beschreibt einen einfachen Rechner, der über die serielle Schnittstelle bedient wird.

```
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;
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.
```

Anhang C – Beispiel für ein Programm mit Objekten:

```
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.
```

Anhang D – Beispiele für die Definition von Konstanten:

```
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;
```

Anhang E – Beispiele für die Deklaration von Typen:

```
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;
```

Anhang F – Beispiele für die Deklaration von Variablen:

```
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;
```

Anhang G – Beispiele für Assembler-Prozeduren:

```
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.
```

Anhang H – Beispiele für Inline-Prozeduren:

```
{
    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.
```

Anhang I – Beispiele für absolute Prozeduren:

```
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.
```

Anhang J – Beispiele für Interrupt-Prozeduren:

```
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.
```

Anhang K – Beispiele für Assembler-Anweisungen:

```
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;
```