Appendix A: The Implemented Language µOberon

This Appendix lists the differences between µOberon and the standard Oberon [1] and may serve as an overview for programmers that already know Oberon.

A.1 Differences Between Oberon and µOberon

The main difference lies in the fact that µOberon does not support record extensions and open array parameters. Moreover there is no garbage collector to release the memory used by objects that are not referenced any more; use a free list instead. On the other side, µOberon introduces two new notations for numbers and characters:

- A binary number may be written as follows: 01001101B, i.e. the trailing letter B identifies a binary number.
- A character may now be written as follows: 32C, i.e. the trailing letter C identifies decimal notation of the ASCII value of the character.

The reason for introducing these two new notations was the unsatisfying situation when writing values for special function registers in assembler sections. The binary notation of a byte allows more transparent programming, i.e. it is more transparent which special function bit is set or not.

Finally, interrupt service procedures and code procedures have a different notation in µOberon, and the trap procedure is a new feature. Refer to the next sections for details.

A.2 Interrupt Service Procedures

In conventional Oberon implementations, procedures that are evoked by an interrupt need to be marked by a plus sign:

```
PROCEDURE +Timer1Overflow;
BEGIN ...
END Timer1Overflow;
```

This causes the registers to be stored upon entry and to be restored upon exit, and leads to an exit from the procedure with a return from interrupt instead of a return from subroutine instruction. After that, they must be dynamically installed by the kernel, i.e. the procedure's address is assigned to the computer's interrupt vector:

```
Kernel.Install(Timer1Overflow, 7)
```

where 7 in this example indicates the vector number.

In µOberon, interrupt service procedures are bound statically to interrupts during the link process. This is done by

```
PROCEDURE (InterruptAddr) MyInterrupt;
BEGIN ...
END MyInterrupt;
```

Modeled on the notation of type bound procedures in Oberon-2, the constant expression in parentheses after the PROCEDURE keyword binds the procedure to the specific interrupt with that address. Note that only one procedure may be defined for a particular interrupt. Otherwise the linker or even the compiler prompts with the corresponding error message. The interrupt address must be an integer number greater than zero, because address 0H is the location where the microcontroller starts execution after reset. Note also that interrupt service procedures must not have any parameters.

In case of an interrupt for which no service procedure is defined, the system ends up in an unpredictable behaviour, because the controller branches to any instruction or even to an operand that will be interpreted as an instruction.

A.3 Trap Procedure

Similar to interrupt service procedures, a trap procedure may be defined that is evoked in case of a runtime trap. The trap procedure is written as an interrupt service procedure with address 0H:

```
PROCEDURE (0) MyTrap(n: SHORTINT);
BEGIN ...
END MyTrap;
```

Unlike interrupt service procedures, the trap procedure must have a single SHORTINT parameter that will contain the number of the runtime trap. There is a list of all possible runtime traps at the end of this chapter.

The trap procedure is called from different locations, e.g. in CASE statements without ELSE branch or on NIL references. But there is not always an explicitly defined trap procedure, so if trap calls are inserted and no trap procedure is defined, an empty procedure, a single return statement is inserted automatically. The linker prompts that with a warning, because correct program behaviour may not be guaranteed in that case.

A.4 The Inline Assembler

There are several notations for code procedures. The assembler of the Oberon system on the Ceres workstation requires an interface module that only consists of procedure headings to be compiled, and the separate assembler generates the code belonging to it. Oberon System 3 allows code procedures:

```
PROCEDURE -Test
   2EH, 04H;
```

Oberon/F expects [code] instead of the minus sign. In Oberon 4 the assembler is evoked as follows:

```
(*$ Dinline.Assemble ... *)
```

The programmer has to code the mnemonics by himself. As long as the inline code procedures aren't too extensive this is a simple way without excessive extension of the language.

Because µOberon is more hardware oriented than conventional Oberon Systems are, there has to be an easier way, i.e. a built-in assembler. Like Turbo Pascal and several

Modula implementations, µOberon provides the ASM instruction. The syntax of assembler sections in µOberon can be found in section *EBNF of µOberon*.

The inline assembler gets evoked at two different positions. Firstly, there is the notation of an assembler section, enclosed by keywords ASM and END, that may occur wherever an ordinary statement may be. The written code will directly be inlined at the current position. Secondly, code procedures and module bodies may be written if there is an ASM keyword instead of BEGIN:

```
PROCEDURE MyCode;
ASM
   INC DPTR
   MOVC A, @A+DPTR
END MyCode;
```

The following points must be taken into account when writing assembler sections:

- The inline assembler may only be evoked if the module SYSTEM is imported because inlined code inherently is system dependent and not portable therefore.
- The semicolon as separation symbol between assembler instructions is optional.
- Comments are written as usual between (* and *).
- The procedure prologue and epilogue are generated automatically. Global as well as local variables and parameters may be accessed as usual, with the restriction that the index of an array must be a constant value. It is even possible to access constants and record fields.
- To reduce the danger of opaque code, labels are only visible within the assembler section they are defined in.
- The inline assembler will insert all 16 bit addresses that stem from labels into the fixup list, so the linker may fix them up correctly. If an absolute 16 bit address is written, e.g. LJMP 8000H, it will not be affected by the linker, so it is possible to call any monitor procedures.
- Labels are not part of the symbol table but are stored in an internal list of the inline assembler. So, it is possible that a label has the same name as a constant for instance, and the mode of the operand determines whether the label or the constant is taken. But it is recommended to avoid such ambiguous situations.
- ACALL and AJMP instructions are used to reduce the code size, i.e. only the lower eleven bits of the new address may be specified, while the upper five bits originate from the current program counter. So it is only possible to branch within the current 2 kB segment. Since absolute addresses are only known after linking, the compiler may not check if the target address lies within the same segment. In such a situation, the assembler prompts with a warning. Moreover the target addresses of these instructions are not fixed up by the linker. So it is not advisable to make use of them.

It is also possible to write interrupt service procedures or the trap procedure in assembler language if the interrupt address or 0 respectively is between the PROCEDURE keyword and the name of the procedure.

A.5 EBNF of µOberon

The Extended Backus-Naur Formalism (EBNF) is a notation to describe the syntax of a language. Brackets [and] denote optionality of the enclosed terms. Braces { and } denote its repetition, possibly zero times. Parentheses (and) group terms. A choice finally is

indicated by a vertical bar |. Terms written in italics differ from the standard Oberon EBNF.

```
BinaryDigit         = "0" | "1".
Digit               = BinaryDigit | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
HexDigit            = Digit | "A" | "B" | "C" | "D" | "E" | "F".
ScaleFactor         = ("E" | "D") ["+" | "-"] Digit {Digit}.
Integer             = BinaryDigit {BinaryDigit} "B" |Digit {Digit}
                      | Digit {HexDigit} "H".
Real                = Digit {Digit} "." {Digit} [ScaleFactor].
Number              = Integer | Real.
Ident               = letter {letter | Digit}.
Character           = Digit {Digit} "C" | Digit {HexDigit} "X".
String              = "'" {char} "'" | '"' {char} '"' |Character.

IdentDef            = Ident ["*"].
Qualident           = [Ident "."] Ident.
ConstantDeclaration = IdentDef "=" ConstExpression.
ConstExpression     = Expression.
TypeDeclaration     = IdentDef "=" Type.
Type                = Qualident | ArrayType | RecordType | PointerType
                      | ProcedureType.
ArrayType           = ARRAY Length {"," Length} OF Type.
Length              = ConstExpression.
RecordType          = RECORD FieldListSequence END.
FieldListSequence   = FieldList {";" FieldList}.
FieldList           = [IdentList ":" Type].
IdentList           = IdentDef {"," IdentDef}.
PointerType         = POINTER TO Type.
ProcedureType       = PROCEDURE [FormalParameters].
VariableDeclaration = IdentList ":" Type.

Designator          = Qualident {"." Ident | "[" ExpressionList "]" | "^" }.
ExpressionList      = Expression {"," Expression}.
Expression          = SimpleExpression [Relation SimpleExpression].
Relation            = "=" | "#" | "<" | "<=" | ">" | ">=" | IN.
SimpleExpression    = ["+"|"-"] Term {AddOperator Term}.
AddOperator         = "+" | "-" | OR.
Term                = Factor {MulOperator Factor}.
MulOperator         = "*" | "/" | DIV | MOD | "&".
Factor              = Number | Character | String | NIL | Set | Designator
                      [ActualParameters] | "(" Expression ")" | "~" Factor.
Set                 = "{" [Element {"," Element}] "}".
Element             = Expression [".." Expression].
ActualParameters    = "(" [ExpressionList] ")".

Statement           = [Assignment | ProcedureCall | IfStatement | CaseStatement
                      | WhileStatement | RepeatStatement |ForStatement
                      | LoopStatement | EXIT | RETURN [Expression]
                      | ASM AsmSection END].
Assignment          = Designator ":=" Expression.
```