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.

```
ProcedureCall           = Designator [ActualParameters].
StatementSequence       = Statement {";" Statement}.
IfStatement             = IF Expression THEN StatementSequence
                          {ELSIF Expression THEN StatementSequence}
                          [ELSE StatementSequence] END.
CaseStatement           = CASE Expression OF Case {"|" Case}
                          [ELSE StatementSequence] END.
Case                    = [CaseLabelList ":" StatementSequence].
CaseLabelList           = CaseLabel {"," CaseLabel}.
CaseLabel               = ConstExpression [".." ConstExpression].
WhileStatement          = WHILE Expression DO StatementSequence END.
RepeatStatement         = REPEAT StatementSequence UNTIL Expression.
ForStatement            = FOR Ident ":=" Expression TO Expression
                          [BY ConstExpression] DO StatementSequence END.
LoopStatement           = LOOP StatementSequence END.

AsmSection              = [AsmCmd {[";"] AsmCmd}].
AsmCmd                  = [[Label ":"] Instruction [Operand {"," Operand} ]].
Label                   = Ident.
Instruction             = Ident.
Operand                 = Rn | "@" Ri | Direct | "#" Data | Bit | "/" Bit
                          | A | AB | DPTR | C | "@" DPTR |"@" A "+" DPTR
                          |"@" A "+" PC | Addr | Rel.
Ri                      = R0 | R1.
Rn                      = Ri | R2 | R3 | R4 | R5 | R6 | R7.
Direct                  = (Integer | Designator) ["+" | "-" Integer].
Data                    = (Integer | Qualident) ["+" | "-" Integer].
Bit                     = (Integer | Qualident) ["+" | "-" Integer].
Addr                    = Label | Integer.
Rel                     = Label | ["-"] Integer.

ProcedureDeclaration    = ProcedureHeading ";" ProcedureBody Ident.
ProcedureHeading        = PROCEDURE ["(" InterruptAddr ")"] IdentDef
                          [FormalParameters].
InterruptAddr           = ConstExpression.
ProcedureBody           = DeclarationSequence
                          [BEGIN StatementSequence | ASM AsmSection] END.
ForwardDeclaration      = PROCEDURE "^" IdentDef [FormalParameters].
DeclarationSequence     = {CONST {ConstantDeclaration ";"}
                          | TYPE {TypeDeclaration ";"}
                          | VAR {VariableDeclaration ";"}}
                          {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
FormalParameters        = "(" [FPSection {";" FPSection}] ")" [":" Qualident].
FPSection               = [VAR] Ident {"," Ident} ":" FormalType.
FormalType              = Qualident | ProcedureType.
ImportList              = IMPORT Import {"," Import} ";".
Import                  = [Ident ":="] Ident.
Module                  = MODULE Ident ";" [ImportList] DeclarationSequence
                          [BEGIN StatementSequence | ASM AsmSection]
                          END Ident ".".
```

A.6 Predefined Procedures

v stands for a variable, x and n for expressions, and T for a type.

Basic Types

Type	*Size [bytes]*	*MIN(T)*	*MAX(T)*
BOOLEAN	1		
CHAR	1	0	255
SET	1	0	7
SHORTINT	1	-128	127
INTEGER	2	-32768	32767
LONGINT	4	-2147483648	2147483647
REAL	4		
LONGREAL	8		
POINTER	2	0	65535

Function Procedures

Name	*Argument type*	*Result type*	*Function*
ABS(x)	numeric type	type of x	absolute value
ODD(x)	integer type	BOOLEAN	x MOD 2 = 1
CAP(x)	CHAR	CHAR	corresponding capital letter
ASH(x)	x, n: integer type	LONGINT	x * 2^n, arithmetic shift
LEN(v,n)	v: array n: integer type	LONGINT	the length of v in dimension n
LEN(v)	array type	LONGINT	LEN(v,0)
MAX(T)	T = basic type	T	max. value of type T
	T = SET	INTEGER	max. element of sets
MIT(T)	T = basic type	T	min. value of type T
	T = SET	INTEGER	0
SIZE(T)	T = any type	integer type	number of bytes required by T

Type Conversion Procedures

Name	*Argument type*	*Result type*	*Function*
ORD(x)	CHAR	INTEGER	ordinal number of x
CHR(x)	integer type	CHAR	character with ordinal number x
SHORT(x)	LONGINT	INTEGER	identity; truncation possible
	INTEGER	SHORTINT	
	LONGREAL	REAL	
LONG(x)	SHORTINT	INTEGER	identity
	INTEGER	LONGINT	
	REAL	LONGREAL	
ENTIER(x)	real type	LONGINT	largest integer not greater than x

Proper Procedures

Name	*Argument type*	*Function*
INC(v)	integer type	v := v + 1
INC(v,x)	integer type	v := v + x
DEC(v)	integer type	v := v - 1
DEC(v,x)	integer type	v := v - x
INCL(v,x)	v: SET x: integer type	v := v + {x}
EXCL(v,x)	v: SET x: integer type	v := v - {x}
COPY(x,v)	x: character array, string v: character array	v := x
NEW(v)	pointer type	allocate v^
HALT(x)	integer constant	terminate program execution

A.7 The Module SYSTEM

v stands for a variable, x, a and n for expressions, and T for a type. SYSTEM exports the type BYTE which is compatible with CHAR and SHORTINT.

Function Procedures

Name	*Argument types*	*Result type*	*Function*
ADR(v)	any	INTEGER	address of variable v
BIT(a,n)	a: LONGINT n: integer type	BOOLEAN	bit n of Mem[a]
PSW(n)	integer constant	BOOLEAN	Program Status Bit n
LSH(x,n)	x: integer type or SET n: -1, 1	type of x	logical shift
ROT(x,n)	x: integer type or SET n: -1, 1	type of x	rotation
VAL(T,x)	T, x: any type	T	x interpreted as of type T

Proper Procedures

Name	*Argument types*	*Function*
GET(a,v)	a: LONGINT v: any basic type	v := ExternalMem[a]
PUT(a,x)	a: LONGINT v: any basic type	ExternalMem[a] := x
IGET(a,v)	a: LONGINT v: any basic type	v := InternalMem[a]
IPUT(a,x)	a: LONGINT v: any basic type	InternalMem[a] := x
CGET(a,v)	a: LONGINT v: any basic type	v := CodeMem[a]
MOVE(s,d,n)	s, d: LONGINT n: integer type	Mem[d]..Mem[d+n-1] := Mem[s]..Mem[s+n-1)
NEW(v,n)	v: any pointer type n: integer type	allocate storage block of n bytes, assign its address to v

A.8 Trap Numbers

1	Command terminated
2	External memory overflow
3	Division by zero
4	Invalid case in CASE statement
5	Stack overflow
6	Invalid index
7	Illegal address (NIL reference)
8	Function procedure without RETURN statement
≥ 9	Programmed HALT

A.9 ASCII Character Set

	0	1	2	3	4	5	6	7
0	NUL	DLE		0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Appendix B: Mcs51 Hardware Overview

This Appendix contains a detailed presentation of the target system. It may be used as a reference when accessing the controller's resources.

B.1 Mcs51 Microcontroller Family

Intel introduced the Mcs51 microcontroller family in 1981 as a successor of the Mcs48, which included the first microcontroller of Intel; the 8048 of 1976. The Mcs51 family is twice to five times faster than the microcontrollers of the Mcs48 family and they supplementary provide Boolean calculations. The Mcs51 microcontrollers have become the industry standard for embedded control. The main characteristics are

- 8 bit CPU optimized for steering and control applications
- Powerful Boolean operations
- Large instruction set
- Internal oscillator, 12 MHz
- 32 bi-directional and individual addressable I/O pins
- Programmable full duplex serial port (UART)
- 5 interrupt sources with 2 priority levels
- Max. 64 kB program store (ROM)
- Max. 64 kB external data store (RAM)
- 128 byte internal RAM
- Two 16 bit timer/counter
- CMOS-version: Power-down and idle-mode
- Compatible to the 8085/88 periphery

According to [18], some of the major manufacturers that produce Mcs51 derivatives are ADM, Atmel, Dallas, Intel, OKI, Philips and Siemens. This wide range makes the Mcs51 the world's most popular microcontroller core. There were 126 million 8051s and variants shipped in 1993. There exist various variants including microcontrollers with integrated ROM or EPROM, 256 instead of 128 byte internal RAM, three 16 bit timers, 6 interrupt sources, analog-to-digital converter, pulse-width modulation, I^2C interface, watchdog timer, automatic multiprocessor communication and so on. The 80537 even provides 16 by 16 bit multiplication or division instructions, but they are not supported by µOberon. There are user programmable and erasable EPROM versions, but beside of being quite expensive, the developer needs a special programming device in order to burn his programs into the internal EPROM. Types with internal ROM may be used like variants without ROM when connecting the external access (EA) pin to ground. The internal ROM will be disabled then.

The main differences between the most popular, ROM-less versions 8031 and 8032 are listed below.

	8031	8032	80C535
internal RAM	128 bytes	256 bytes	256 bytes
timer/counter	2	3	4
interrupt sources	5	6	12
	equal to 8051, but has no internal ROM	equal to 8052, but has no internal ROM	8 channel 8 bit ADC

The following sections cover the two basic members of the Mcs51 microcontroller family, the 8031 and 8032. All other types are based on them, so code written for an 8031 will also work on an 80C535. The 80C535 is only one example of a sophisticated derivative in the Mcs51 family.

B.2 Architecture

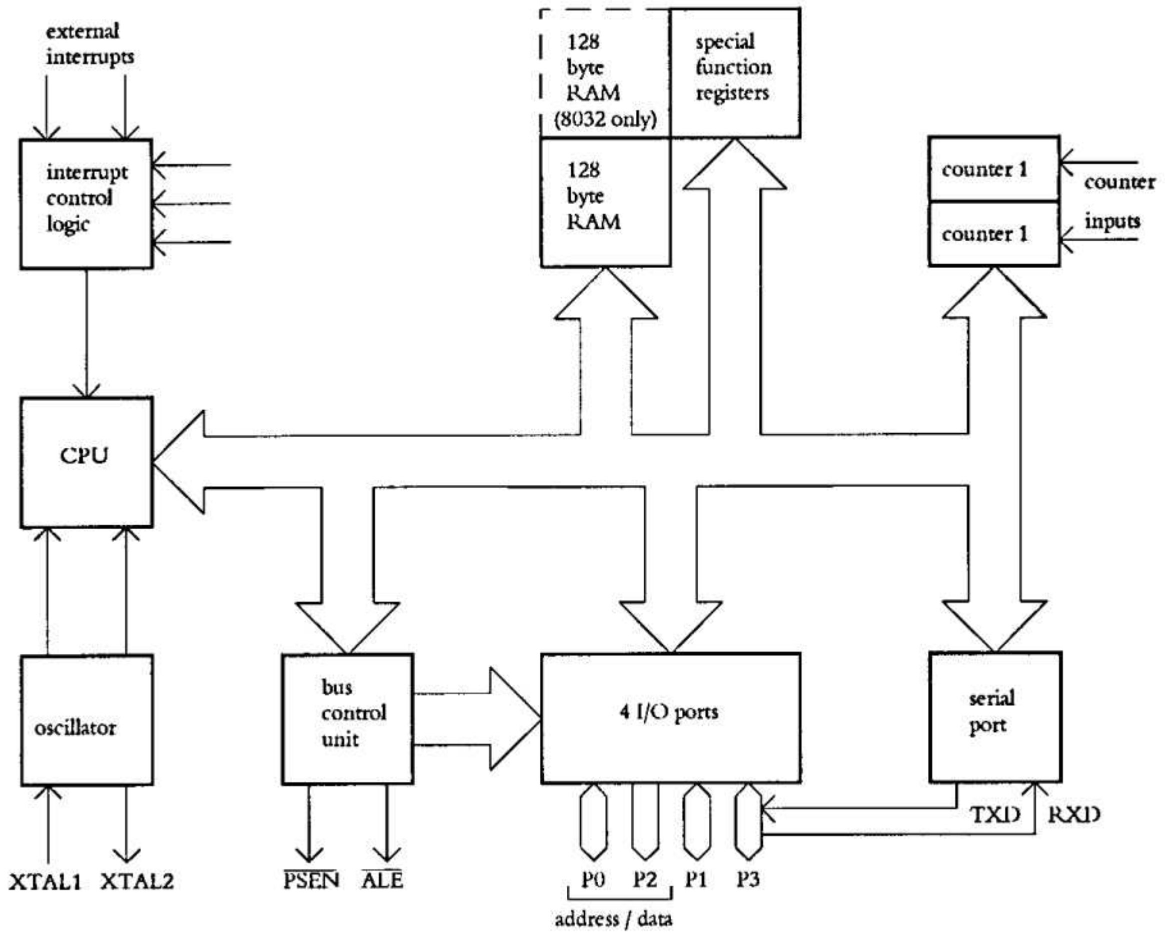


Figure 15 The architecture of Mcs51 microcontrollers.

Pin Configuration

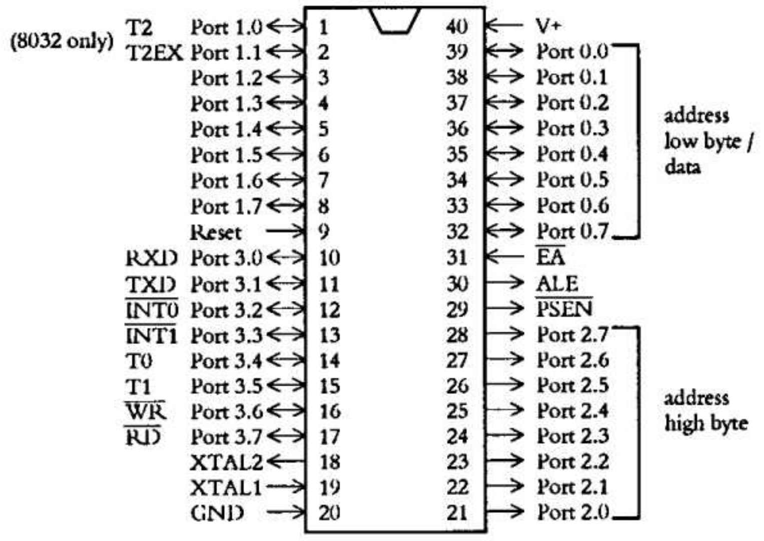


Figure 16 Pin configuration of Mcs51 microcontrollers.

RXD and TXD are used for the serial interface, INT0 and INT1 are the pins that may be used to cause an interruption of the main program, T0, T1 and T2 are used to count external events or to measure pulse width.

RD and WR provide the read and write signal for an external RAM and are connected to output enable (OE) and write enable (WE) respectively. As mentioned above, the Mcs51 separates code and data memory. Usually the code memory is an EPROM, and a static RAM is used for the data memory. In this configuration the PSEN pin enables the EPROM, and port pins 3.6 and 3.7, WR and RD, are used for RAM accesses. If no external RAM is present, these port pins may be used for other I/O functions.

XTAL1 and XTAL2 are connected to the external quartz crystal. ALE is the address latch enable signal. Since the address low byte and the data are multiplexed in order to reduce the pin count which is an essential criteria of production costs, the address low byte has to be stored in the latch like a 74'373 or '573 during the access.

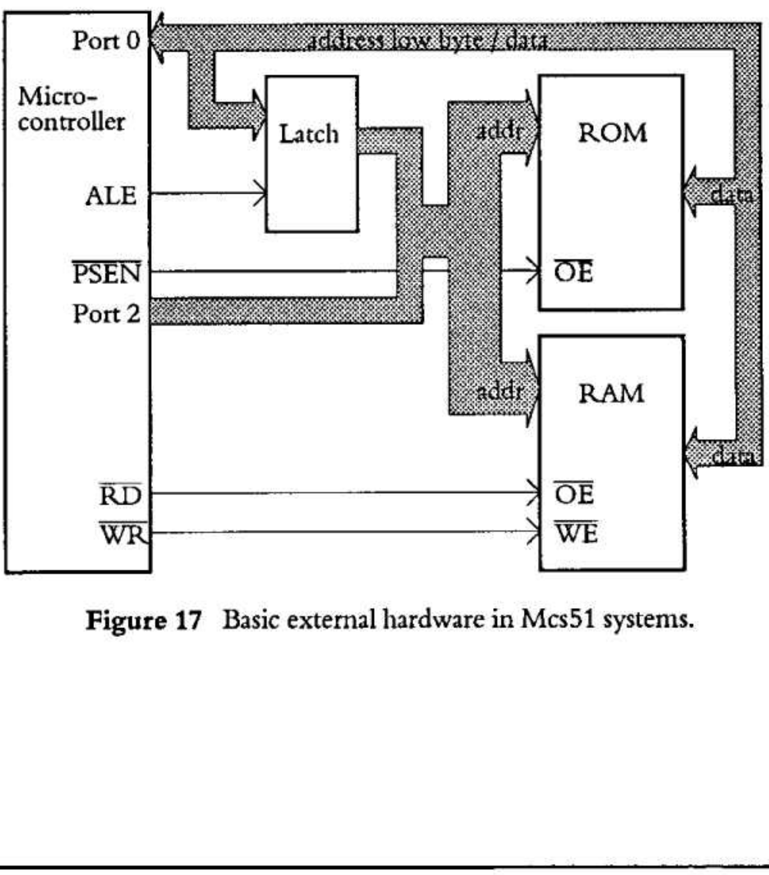


Figure 17 Basic external hardware in Mcs51 systems.

PSEN is the program store enable signal that enables the program store while reading an instruction. Because of this pin Mcs51 microcontrollers differ from the von Neumann, or Princeton architecture since the code and the data memory space are separated. It is to mention that the main advantage of Harvard architectures concerning pipelining, i.e. simultaneous operand and next instruction fetch, can certainly not be used; this would require separate address and data busses. This is not the idea of a simple 8 bit microcontroller.

In order to map the external code and data address space together, e.g. to enable a monitor program to download code from a host system, the external hardware showed in figure 18 is necessary.

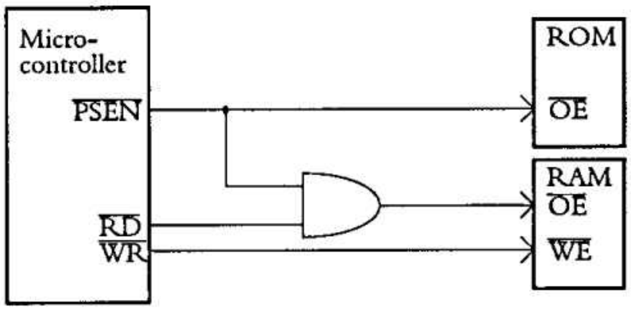


Figure 18 External hardware to unite the external memory space.

Since the PSEN and RD signals both are negative logic, the output of the RAM will be active if one of PSEN and RD is active.

When there is no external RAM and no need for serial communication neither for external interrupts nor for counter events, the pins of ports 1 and 3 may be used either as input or as output without the need of a data direction register. An input pin must be written to logic high prior to the readout because otherwise a short circuit could result when the internal flip flop is in low state and the external source tries to pull up the voltage level.

Moreover the port structure is to be considered. Some instructions read the content of the flip flop, others read the port pin state. If a port pin shall act as an input, a logical 1 has to be in the port latch. Instructions that read the flip flop and not directly the pin are instructions that may read-modify-write the pin's state. These are ANL, ORL, XRL, JBC, CPL, INC, DEC, DJNZ, MOV Px.y,C, CLR Px.y, SETB Px.y, as long as they deal with port addresses. It is not obvious that the last three instructions do this as well. If a port bit has to be set or reset, the CPU will not do this separately. The CPU reads the whole eight bits, modifies only the addressed one and writes back the result as a byte. This is done to eliminate misinterpretations of the voltage level, e.g. if a port bit drives a transistor. In this case the base voltage could sink so that the CPU may read a logical 0 instead of a logical 1.

The following instructions release the port pin latch: MOV A,Px; MOV Rn,Px; MOV Px,direct; PUSH Px; XCH A,Px; ADD A,Px; ADDC A,Px; SUBB A,Px; CJNE A,Px,rel; JB Px.y,rel; JNB Px.y,rel.

B.3 Code Memory

After a reset, all port bits are set so they may work as inputs, the stack pointer is initialized to 07H, and all special function registers are initialized by the values shown in the table in the next section. The internal RAM as well as the serial buffer (SBUF) special function register will remain unchanged. The program counter PC points to the memory location 0000H and the CPU reads and executes the first instruction. Usually this is a jump instruction to the main program because the following addresses in the code memory are used for special purposes. If interrupts are not used, the main program may start at address 0000H.

002BH	Timer 2 overflow / External interrupt 2	TF2+EXF2	8032 only
0023H	Serial port interrupt (receive/transmit)	RI+TI	
001BH	Overflow counter/timer 1	TF1	
0013H	External interrupt 1 (INT1)	IE1	
000BH	Overflow counter/timer 0	TF0	
0003H	External interrupt 0 (INT0)	IE0	
0000H	Reset		

Since most interrupt routines are longer than eight bytes, the code memory locations used by interrupts usually contain a call or, more conveniently, a jump instruction to the corresponding routine, though a jump is better since the limited stack is saved this way. An interrupt routine must be finished by a RETI instruction; this enables further interrupts by clearing the interrupt busy flag. Note that Mcs51 microcontrollers do not save any registers automatically, so the programmer has to save the used registers, like the program status word.

More sophisticated members of the Mcs51 family provide more interrupt sources, so more code addresses are reserved. The 80C535 for instance uses code memory addresses up to 006BH for interrupts.

B.4 Data Memory

The address organization of the RAM is more complex because there is a distinction between the internal and an optional external RAM.

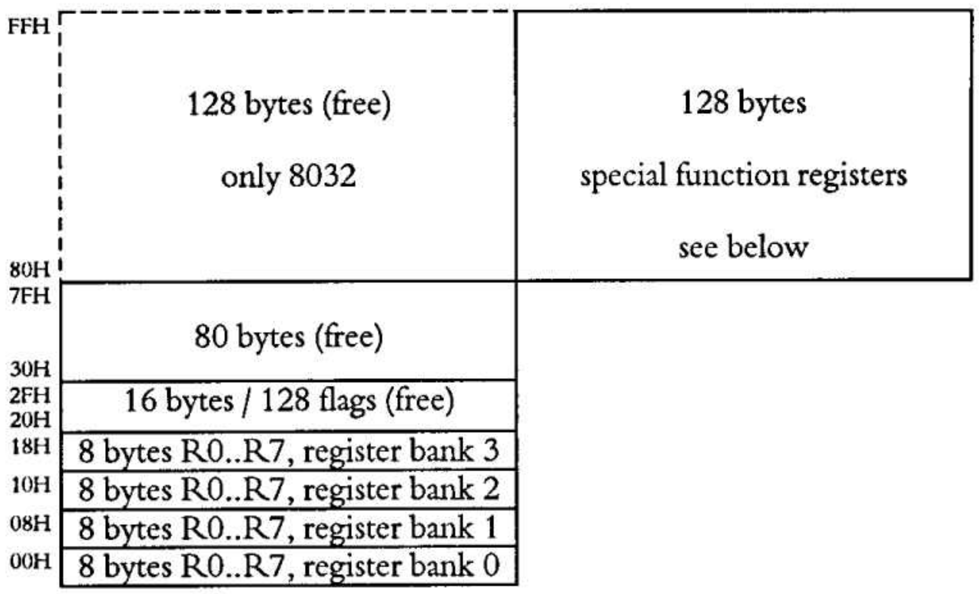

FFH
128 bytes (free)
only 8032
128 bytes
special function registers
see below
80H
7FH
80 bytes (free)
30H
2FH
16 bytes / 128 flags (free)
20H
18H
8 bytes R0..R7, register bank 3
10H
8 bytes R0..R7, register bank 2
08H
8 bytes R0..R7, register bank 1
00H
8 bytes R0..R7, register bank 0

Figure 19 Organization of the internal memory.

The 128 bytes from 00H to 7FH may be addressed directly or indirectly using the pointer register R0 or R1. Addresses 00H to 1FH contain four versions of the register bank. The current register bank is selected by bits 3 and 4 of the program status word. This provides a fast context switch; the registers used in an interrupt service routine don't have to be pushed onto the stack. Using only one register bank, the unused register bank areas may be used for other purposes. Addresses 20H to 2FH contain 16 bytes that are bit addressable each. These bits are numbered from 0 to 127. For example, bit number 0 is the least significant bit of the byte at address 20H.

On the 8032, the upper half of the internal RAM is divided into two parallel parts. While the left side, the extra 128 bytes RAM which is only available on a 8032, only may be accessed using indirect addressing, the special function registers at the right side are solely directly addressable.

B.5 Special Function Registers

This section gives a general idea of some special function registers. It is not intended to be used as a complete reference but rather as an introduction to the use of registers in order to operate with on-chip hardware like timers, interfaces and so on.

In order to access this hardware, one has to write to or read from the so-called special function registers. So there is no need for special instructions reflecting this hardware, and later extended versions of microcontrollers providing even more on-chip hardware are still object code compatible with the 8031; the extra hardware is made accessible by adding further special function registers.

			bit addressable	*Initial value after reset*	*Types*
0F0H	B register	B	✓	0000'0000B	
0E0H	Accumulator	ACC	✓	0000'0000B	
0D0H	Program status word	PSW	✓	0000'0000B	
0CDH	Timer 2 high byte	TH2		0000'0000B	8032 only
0CCH	Timer 2 low byte	TL2		0000'0000B	8032 only
0CBH	Capture reg high byte	RCAP2H		0000'0000B	8032 only
0CAH	Capture reg low byte	RCAP2L		0000'0000B	8032 only
0C8H	Timer 2 control	T2CON	✓	0000'0000B	8032 only
0B8H	Interrupt priorities	IP	✓	xxx0'0000B	8031
				xx00'0000B	8032
0B0H	Port 3	P3	✓	1111'1111B	
0A8H	Interrupt enables	IE	✓	0xx0'0000B	8031
				0x00'0000B	8032
0A0H	Port 2	P2	✓	1111'1111B	
099H	Serial buffer	SBUF		xxxx'xxxxB	
098H	Serial control	SCON	✓	0000'0000B	
090H	Port 1	P1	✓	1111'1111B	
08DH	Timer 1 high byte	TH1		0000'0000B	
08CH	Timer 0 high byte	TH0		0000'0000B	
08BH	Timer 1 low byte	TL1		0000'0000B	
08AH	Timer 0 low byte	TL0		0000'0000B	
089H	Timer modes	TMOD		0000'0000B	
088H	Timer control	TCON	✓	0000'0000B	
087H	Processor control	PCON		0xxx'xxxxB	NMOS
				0xxx'0000B	CMOS
083H	Data pointer high byte	DPH		0000'0000B	
082H	Data pointer low byte	DPL		0000'0000B	
081H	Stack pointer	SP		0000'0111B	
080H	Port 0	P0	✓	1111'1111B	

The table shows that special function registers are bit addressable if the three least significant bits of their addresses are cleared. The bit number is calculated as the sum of the register address plus the bit number. So, bit number 087H is the most significant bit of port 0.

Program Status Word PSW

	7	6	5	4	3	2	1	0
0D0H	Carry CY	OV from bit 4 AC	User flag F0	RegBank select 1 RS1	RegBank select 0 RS0	OV from bit 6 OV	free	Parity $\overline{\text{even}}$/odd P
Reset :	0	0	0	0	0	0	0	0
Bit number :	0D7H	0D6H	0D5H	0D4H	0D3H	0D2H	0D1H	0D0H

The carry bit CY is the accumulator for Boolean operations. The Carry may also be regarded as an overflow from bit 7 when dealing with unsigned numbers. - The carry is cleared after a division by zero.

Overflow from bit 4, the auxiliary carry (AC), may be used for BCD additions. The user flag F0 is for free general use. RS1 and RS0 select the current register bank, i.e. whether the registers R0 to R7 should be located at 00H to 07H, 08H to 0FH, 10H to 17H or at 18H to 1FH. This allows a fast context switch. - µOberon always uses bank 0 and does not switch the register bank. OV indicates an overflow from bit 6 when calculating with signed numbers. OV will also be set if the result of a multiplication is longer than one byte, or after a division by zero. Bit 1 is free for general use, and the parity bit P indicates whether there is an even (P reset) or an odd (P set) number of set bits in the accumulator, and is set/reset after each instruction.

The following instructions may influence the content of the program status word (x: may be influenced; 1: will be set; 0: will be reset):

	CY	OV	AC
ADD	x	x	x
ADDC	x	x	x
SUBB	x	x	x
MUL	0	x	
DIV	0	x	
DA	x		
RRC	x		
RLC	x		
SETB C	1		
CLR C	0		
CPL C	x		
ANL C,bit	x		
ANL C,/bit	x		
ORL C,bit	x		
ORL C,/bit	x		
MOV C,bit	x		
CJNE	x		

plus each instruction writing to address 0D0H. Also instructions that change the value of the accumulator may change the parity bit in the program status word. Note that INC and DEC instructions do not affect the CY, OV and AC bits, so it may be necessary to use ADD #1 or CLR C SUBB #1 instead. Note that it is important to clear the carry prior to the subtraction; refer to the instruction set summary.

In common microprocessors there is a zero bit in the program status word or a condition code register that indicates whether the accumulator is equal to zero or not. Mcs51-microcontrollers allow this testing by using JZ and JNZ instructions.

Stack Pointer SP

The call of a subroutine causes the return address to be pushed onto the stack, so the processor knows where to go back after having processed the subroutine. First the stack pointer is incremented by one, and the least significant byte of the return address, the address of the next instruction byte in the code memory, is written into the memory cell where SP points to. After that, SP is incremented once again and the most significant byte is written to the new location.

So SP always points to the last used memory location and will be preincremented. Vice versa SP is postdecremented after POP. The initial value of SP is 07H, so the first byte pushed on the stack will be situated immediately after register bank 0, at R0 of register bank 1. On a 8031, the stack pointer goes back to 00H after exceeding 7FH. On a 8032 on the other hand, the stack grows further towards 0FFH, and then goes back to 00H. Note that in this case, the upper 128 bytes are not the special functions registers, but a space of internal memory that is only accessible indirectly.

Processor Control PCON

	7	6	5	4	3	2	1	0
087H	Ser. port mode SMOD				General Flag 1 GF1	General Flag 0 GF0	Power down PD	Idle mode IDL
Reset :	0	x	x	x	0	0	0	0

Not bit addressable (CMOS only)

CMOS versions of the microcontrollers consume less current than NMOS versions. Moreover they provide two modes called *idle* and *power-down*. Note that the lower four bits of PCON are not available on NMOS types.

In idle mode, the CPU clock is switched off, instruction execution is stopped, the port pins hold their actual values, only the RAM and the other peripheral functions like timers and the serial port are still working. The other registers hold their current values. Power consumption is reduced to 20%. Only an interrupt or a hardware reset may wake up the microcontroller by clearing the IDL bit. In the case of an interrupt, the IDL bit will be automatically reset prior to the entry of the interrupt routine. If the microcontroller was waked up by a reset, it may happen that two or three instructions following the instruction that evoked the idle mode are executed. The internal RAM access is prohibited in this time, but port pin changes are possible. So it is wise to write three NOP instructions thereafter. - ALE and PSEN pins are both high in idle mode.

The power-down mode turns off all chip activities by stopping the oscillator. Only the content of the internal RAM is conserved, and the port pins hold their current level. In this mode the power consumption is less than 10 µA. Only a reset restarts the microcontroller. During power-down mode, the voltage supply may be reduced from 5 V down to 2 V without losing the internal RAM content. - ALE and PSEN pins are both low in power-down mode.

This modes are evoked by setting the corresponding bits IDL and PD, for example by the instruction OR PCON,#1 or #2 respectively. If both bits are set simultaneously, the power-down mode is evoked.

SMOD is used to define the baud rate for the serial port, see below. SMOD=1 doubles the baud rate. GF0 and GF1 are flags for general purpose and don't have special functions.

They may be used to indicate to an interrupt routine whether the interrupt occurred during normal execution or in idle mode.

Interrupt Enable Register IE

	7	6	5	4	3	2	1	0
0A8H	General enable EA		Timer 2 enable ET2	Serial enable ES	Timer 1 enable ET1	INT 1 enable EX1	Timer 0 enable ET0	INT 0 enable EX0
Reset:	0	x	0	0	0	0	0	0
Bit number:	0AFH	0AEH	0ADH (8032 only)	0ACH	0ABH	0AAH	0A9H	0A8H

There are several interrupt sources. An external interrupt is achieved by connecting one of the low active INT pins to ground. Prior to that the programmer can choose if an external interrupt is falling edge or negative level triggered; see TCON register. Moreover interrupts can be triggered by an overflow of one of the internal timers or by the serial interface. The latter one can be either a receiver or a transmitter interrupt. The serial interrupt routine is at the same address and has to be distinguished using the RI or TI bits in SCON. In order to enable interrupts IE.7 has to be set. Only then the individual enabling of interrupt sources has an effect.

Interrupt Priority Register IP

	7	6	5	4	3	2	1	0
0B8H			Timer 2 priority PT2	Serial Int priority PS	Timer 1 priority PT1	INT 1 priority PX1	Timer 0 priority PT0	INT 0 priority PX0
Reset:	x	x	0	0	0	0	0	0
Bit number:	0BFH	0BEH	0BDH (8032 only)	0BCH	0BBH	0BAH	0B9H	0B8H

There are two priority levels. Only an interrupt of a higher level, with priority bit set, may interrupt another interrupt presently being processed. The internal processing of interrupts makes an inherent priority sequence. That sequence is in use whenever two interrupts of the same priority occur at the same time:

External interrupt 0	INT0		highest priority
Overflow timer 0	TF0		
External interrupt 1	INT1		
Overflow timer 1	TF1		
Serial interface	RI/TI		
Timer 2 / External interrupt 2	TF2/EXF2	8032 only	lowest priority

Timer Mode Register TMOD

	7	6	5	4	3	2	1	0
089H	GATE	Counter/ Timer $C/\overline{T}$	Mode 1 M1	Mode 0 M0	GATE	Counter/ Timer $C/\overline{T}$	Mode 1 M1	Mode 0 M0
Reset:	0	0	0	0	0	0	0	0
Not bit addressable	Timer 1				Timer 0			

GATE has to be cleared in order to enable the timer. If GATE is set, the timer/counter will run only if the INT pin is high. This can be used to measure pulse width: Each time

INT is low, use the JB instruction, read the timer registers. - The figure below illustrates the function of the timer/counter mechanism.

C/T selects whether it should act as a timer (C/T=0) with the system clock divided by 12, so 1 MHz when using a 12 MHz quartz crystal, or as a counter of external events (C/T=1). In counter mode, the timer registers are incremented after each falling edge at the corresponding pin T0 and T1. The maximal counting frequency is 500 kHz using a 12 MHz quartz crystal. The falling edge has to be shorter than one clock cycle.

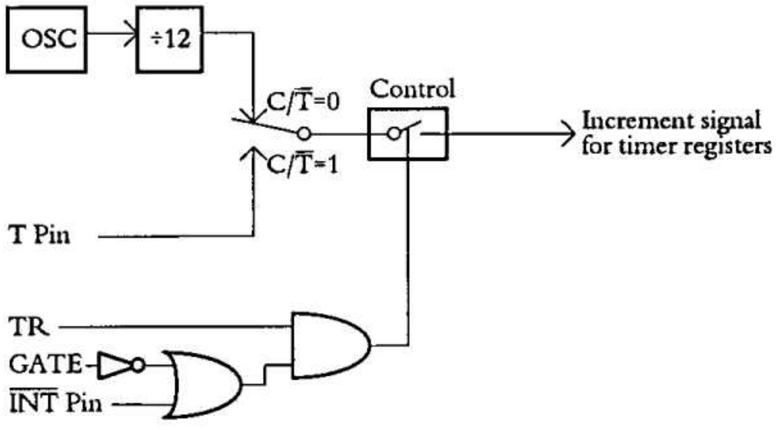


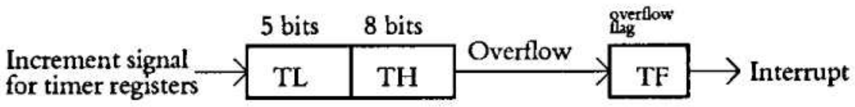
Figure 20 Clock select for timers and counters.

M1 and M0 finally select the operating mode. Each timer is equipped with two 8 bit registers TH and TL.

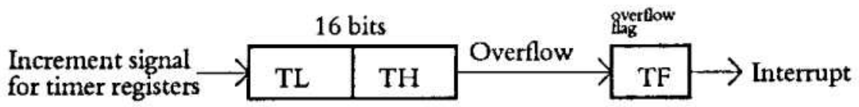
M1	M0	Mode	
0	0	0	13 bit timer/counter mode Timer low byte (TL) serves as a 5 bit prescaler
0	1	1	16 bit timer/counter mode TH and TL are cascaded and there is no prescaler
1	0	2	8 bit auto reload timer/counter mode TH holds a value which will be reloaded into TL after an overflow
1	1	3	Two 8 bit timer/counter mode Timer 0 : ▸ TL0 is an 8 bit timer/counter controlled by the timer 0 control bits ▸ TH0 is an 8 bit timer only controlled by the timer 1 control bits Timer 1 is idle when set to this operating mode

In mode 3, timer 1 will be stopped as if TR1 is reset. Timer 1 may still be used in another mode when timer 0 is in mode 3, but may not cause an interrupt. This mode has therefore only effect to timer 0. So, mode 3 provides three 8 bit counters. To start or to stop Timer 1 in this mode, it has to be set to mode 3 or to another mode respectively.

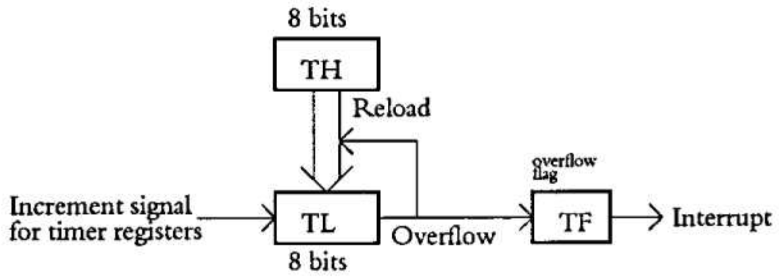
Mode 0



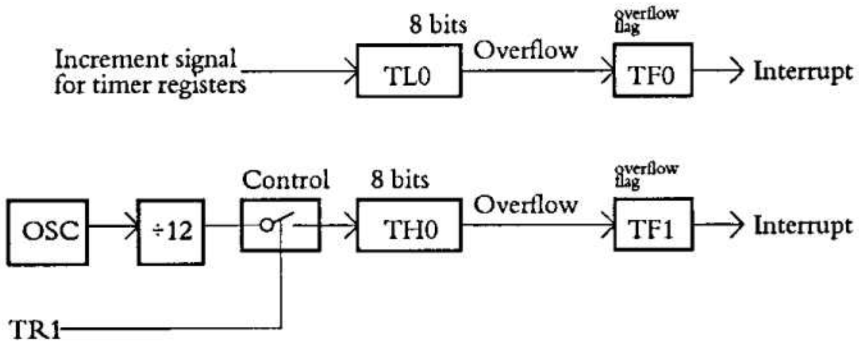
Mode 1



Mode 2



Mode 3



Timer Control Register TCON

	7	6	5	4	3	2	1	0
088H	Timer 1 overflow TF1	Timer 1 start TR1	Timer 0 overflow TF0	Timer 0 start TR0	INT 1 IE1	INT 1 edge/level IT1	INT 0 IE0	INT 0 edge/level IT0
Reset :	0	0	0	0	0	0	0	0
Bit number :	08FH	08EH	08DH	08CH	08BH	08AH	089H	088H

Although this register is called timer control register it contains some bits concerning interrupts as well.

The TF bits are set by the hardware in the case of an overflow of the corresponding timer. After branching to the interrupt routine the hardware clears the bits again. The TR bits are set and reset by the user in order to start and stop the timer. The IE bits are, similar to the TF bits, set by the hardware when an external interrupt edge is detected and, if edge triggered only, reset by hardware while jumping to the interrupt routine. If the external interrupt is level sensitive, the IE bits have to be cleared by software. IT selects whether the external interrupt is edge (IT=1) or level sensitive (IT=0). Note that in edge triggered mode, the edge should not take more than 1 µs assuming a 12 MHz quartz crystal.

To finish an interrupt routine properly it is necessary to use the RETI instruction. This will, in contrast to RET, reset the corresponding interrupt busy bits. Further interrupts are served only after RETI. It is also possible to invoke all interrupts by software by setting the corresponding bits.

Timer 2 Control Register T2CON

Note that this register only exists on a 8032 and is different on the 80C535.

	7	6	5	4	3	2	1	0
0C8H	Timer 2 overflow TF2	External flag EXF2	Receive clock RCLK	Transmit clock TCLK	Enable T2EX pi EXEN2	Timer 2 start TR2	Counter/ timer $C/\overline{T2}$	Capture/ reload flag $CP/\overline{RL2}$
Reset :	0	0	0	0	0	0	0	0
Bit number :	0CFH	0CEH	0CDH	0CCH	0CBH	0CAH	0C9H	0C8H

Timer 2 is a 16 bit timer/counter, too. It may work in the following modes:

RCLK	TCLK	CP/RL2	
0	0	0	16 bit auto reload
0	0	1	16 bit capture
1	1	x	baud rate generator

If RCLK or TCLK is set, the TF2 bit won't be set at an overflow. TF2 has to be reset by the software. On RCLK set, a timer 2 overflow will clock the receiver in mode 1 and 3. If RCLK is reset, this clock is taken from timer 1 overflow events. TCLK is the same for the transmitter clock. On EXEN2 set, a capture or reload of counter 2 is allowed at a negative edge at T2EX pin, assumed timer 2 does not clock the serial port. The TR2 and C/T2 bits are similar to those of timers 0 and 1.

CP/RL2 selects capture (CP/RL2=1) or reload (CP/RL2=1) mode. In capture mode with EXEN2 reset, timer 2 is a 16 bit timer/counter that sets the TF2 bit at an overflow, which may cause an interrupt. On EXEN2 set, the counter has a further feature: If there is a 1-to-0 transition on T2EX pin, the actual content of timer 2, high and low byte, is written to RCAP2H and RCAP2L. At the same time the EXF2 bit is set, which may also cause an interrupt. EXF2 has to be reset by the software. - So the interrupt routine may see in T2CON which type of interrupt it was, since timer 2 overflow and external interrupt 2 share the same interrupt address 002BH.

In auto reload mode, EXEN2 also selects two options. If EXEN2=0, a timer 2 overflow not only causes the set of TF2, but a reload of the counter with the values stored at RCAP2H and RCAP2L. On EXEN2 set, TF2 is set and the counter is reloaded at an overflow, and a negative edge on the T2EX pin will also reload the counter, and the EXF2 bit will be set.

Serial Control Register SCON

	7	6	5	4	3	2	1	0
098H	Serial mode 0 SM0	Serial mode 1 SM1	Serial mode 2 SM2	Receptio enable REN	9th bit transmit TB8	9th bit receive RB8	Transmit interrupt TI	Receive interrupt RI
Reset :	0	0	0	0	0	0	0	0
Bit number :	09FH	09EH	09DH	09CH	09BH	09AH	099H	098H

This register serves for setting and controlling the serial interface.

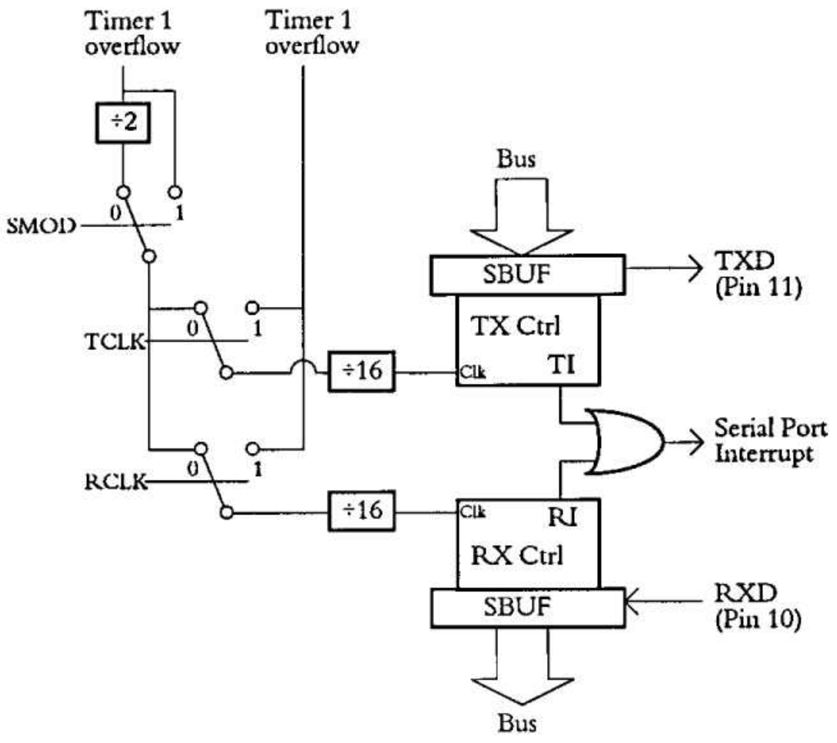


Figure 21 Serial interface architecture.

SMOD is bit 7 in register PCON and selects a prescaler. So if SMOD is set, the baud rate is overflow rate divided by 16, otherwise divided by 32. It is not possible to generate exactly 4800 baud when using a 12 MHz quartz crystal; for this purpose it is necessary to take an 11.0592 MHz quartz crystal. But the resulting 4807.69 bits per second will also work in most cases. The 80C535 is capable to generate exactly 4800 and 9600 baud with a 12 MHz quartz crystal. Bits SM0 and SM1 specify the serial port mode:

SM0	SM1	Mode		Baud rate	
0	0	0	shift register	$f_{osc}/12$	
0	1	1	8 bit UART	variable	full duplex
1	0	2	9 bit UART	$f_{osc}/64$ or $f_{osc}/32$	full duplex
1	0	3	9 bit UART	variable	full duplex

Bit REN enables serial reception and has to be set and reset by the software. TB8 is the 9th data bit that will be transmitted in modes 2 and 3 and may be set or reset by software. RB8 is the received 9th data bit in modes 2 and 3. In mode 1 with cleared SM2, RB8 is the received stop bit. In mode 0, RB8 is not used. TI is the transmit interrupt flag that will be set by the hardware at the end of the 8th bit time in mode 0, or at the beginning of the stop bit in the other modes. This bit has to be cleared by software. In order not to generate a transmit interrupt too early this bit has to be cleared prior to the first transmit.

The 9th data bit may be used for multiprocessor communication. A set 9th bit identifies the transmitted byte as an address, a reset 9th bit identifies a data byte for the previous addressed coprocessor. This way it is theoretically possible to communicate with 256 coprocessors.

RI is the receive interrupt flag that will be set by the hardware at the end of the 8th bit time in mode 0, or halfway through the stop bit time in the other modes. Exceptions: see description of the SM2 bit. This bit has to be cleared by software in order to indicate that the received byte is fetched, i.e. the receive buffer is read.

TI and RI are, assuming that IE.4 (ES) of the interrupt enable register is set, evoking the same interrupt, so the serial interrupt handler has to find out itself whether it was a transmit or receive interrupt. In order to send or receive data from the serial port the microcontroller has an internal shift register called SBUF at address 099H. In fact there are two of them, the receive and the transmit buffer, the corresponding one is selected by the kind of the instruction that accesses the register, a read or a write instruction respectively. This allows full duplex operation. The receiver is buffered; it is possible to receive the next byte while the previous one is still in SBUF. But this byte will be lost if it is not yet fetched after having received the next one. Note that the receiving byte is not shifted in the SBUF register; it will be written into SBUF only after receiving it as a whole. Bytes to transmit aren't shifted in SBUF.

Writing to SBUF starts the transmission. The baud rate is taken either from timer 1 or 2 (only 8032). The 8032 has even the possibility to use different transmit and receive speeds. A bit to be received is headed by a start bit, and the internal 1-to-0 transition detector synchronizes the receiver clock. So a falling edge starts receiving.

Mode 0

This is the only synchronous mode. The RXD pin is used to receive and transmit the data, while TXD clocks the external shift register with $f_{osc}/12$ in each case. Eight bits are received or transmitted respectively, the least significant bit first.

The transmission is started by writing a value into SBUF. A transmit interrupt, TI flag is set, indicates that the transmission has finished. Using an external serial-in, parallel-out shift register, for instance a 74'595, this provides a further output port.

In order to receive data, RI has to be reset and REN set. This clocks eight data bits into SBUF. So there is a further input port. In this mode, SM2 should be cleared.

Mode 1

Each time, ten bits are received or transmitted respectively. This mode works full duplex using RXD and TXD pins. Since in idle position, both pins are supposed to be high level, a transmission starts with a low start bit, the stop bit is high level.

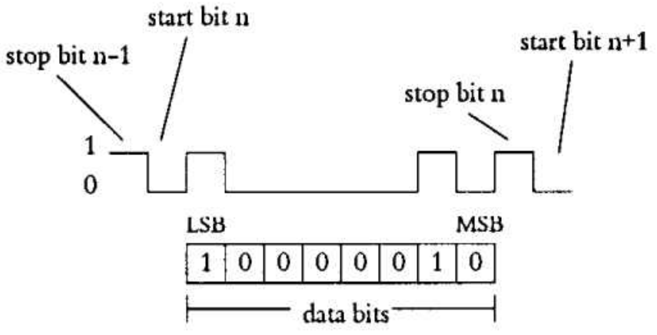


Figure 22 Data for the serial interface.

The baud rate in this mode is calculated by

$$\frac{2^{SMOD} \cdot (\text{overflowrate})}{32}$$

Receiving is started when REN is set and a falling edge is detected at the RXD pin. After having received a byte, the received stop bit is copied into RB8 and the RI flag is set, but only if RI was reset, so the previous interrupt was processed, and SM2 is reset or the stop bit was high level. Otherwise the received byte is ignored. If SM2 is set in this mode, RI will not be activated if not a valid stop bit was received.

Modes 2 and 3

Each time, eleven bits are received or transmitted respectively: A start bit (0), eight data bits, a programmable 9th bit and a stop bit (1). The received 9th bit is written to RB8, the 9th bit to transmit is taken from TB8. The rest is analogous to mode 1.

Bit SM2 enables the multiprocessor communication feature in these modes. A set SM2 will cause RI not to be activated if the received 9th data bit (RB8) is cleared. In multiprocessor communication mode, a set 9th data bit indicates that the just received byte is an address byte that addresses a particular coprocessor. On SM2 reset, there will be an interrupt request after each receive.

This two modes are mainly used for multiprocessor communication. Each processor has set SM2 and REN bits. In order to address a particular processor, the address is written to SBUF after having set TB8. The following transmission will cause an interrupt in each connected microcontroller. The interrupt service routine checks if it is the own address, and the addressed processor clears SM2 and RI, so that the following data bytes, with cleared 9th bit, are received.

Appendix C: Mcs51 Instruction Set Summary

This Appendix lists the entire Mcs51 instruction set and explains the possible addressing modes. µOberon allows code inlining, so this list may be useful for low-level programming.

C.1 Addressing Modes

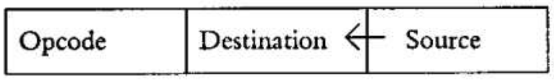


Figure 23 Instruction format of Mcs51 microcontrollers.

The Mcs51 family offers five different addressing modes. The general format is showed in figure 23. The only exception is the MOV direct,direct instruction, where the source address is between the opcode byte and the destination address. But this will not affect the programmer because the inline assembler swaps the operands here.

Register Addressing MOV ..Rn..

Rn may be one of the general purpose registers R0 to R7 of the current register bank, as selected in the program status word. Bits 0 to 2 of the opcode specify the number of the desired register.

Direct Addressing MOV ..direct..

The direct address represents the location of the operand in the internal RAM. This is the only way to access the special function registers. An address lower than 080H will cause an access to the lower 128 bytes of internal memory, where the register banks and the 128 user flags are located, and addresses from 080H to 0FFH are used to access the special function registers.

Register-Indirect Addressing MOV ..@Ri..

Register R0 or R1, one of the two pointer registers, contains the address 00H to FFH of the desired operand. Either the internal or the external RAM is accessed, depending on the instruction mnemonic. Register-indirect addressing is characterized by the @ symbol. So, the instruction MOV A,@R0 loads the content of the memory location pointed to by R0 of the current register bank into the accumulator.

Register-indirect addressing is the only way to access the upper 128 bytes of the internal RAM of 8032 derivatives. This addressing does not access the special function registers that share the same memory locations from addresses 080H to 0FFH therefore.

The PUSH and POP instruction also make use of this way of addressing, i.e. the stack won't overwrite the special function registers if it grows into the upper half of the internal RAM. A register-indirect access to the upper 128 bytes on an 8031, which only provides 128 bytes internal RAM, will return undefined values.

To access the whole 64 kB of external data or code memory space the 16 bit data pointer DPTR is being used.

Immediate Addressing   MOV ..,#10

The operand is specified as a constant value.

Indirect-Offset Addressing   MOVC A,@A+DPTR or @A+PC

This addressing mode is useful when processing tables. The address of the desired byte is the sum of the content of a 16 bit base register plus the content of the accumulator. The base register may either be the data pointer DPTR or the program counter PC.

Bit Addressing   SETB C

Operands for the Boolean processor are bits. There are 256 different bits numbered from 0 to 255, some of them with special functions like the carry bit, the accumulator of the Boolean processor, bits of bit addressable special function registers, or flags for general use.

Bit numbers from 0 to 127 are flags for general use located in the internal RAM at addresses 020H to 02FH, numbers from 128 upwards are bits of bit addressable special function registers.

Relative Addressing   SJMP rel

This addressing mode is for conditional branches or short jumps (SJMP). The instructions branch relative to the current content of the program counter PC. Note that the PC always points to the next instruction to be executed; the PC is incremented immediately after each instruction fetch. So, the effective address is relative to the address of the instruction following the jump instruction.

C.2 Instruction Set

The list shows each instruction with all possible addressing modes. Not every mode is applicable to each instruction; the instruction set is rather non-orthogonal.

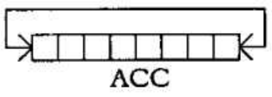
The mnemonics and the operands are listed in the first column. The second column describes the operation executed by the instruction. Brackets [and] denote the content of the operand named by the designator, so [Rn] is the content of register Rn, and [[Ri]] is the content of the memory cell pointed to by register Ri.

The last two columns finally show the number of code bytes and the number of CPU clock cycles, which are 1/12 of the quartz crystal frequency, e.g. one microsecond using a 12 MHz quartz crystal, used by the instruction.

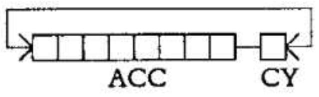
Notation

Rn	may be one of the eight registers R0 .. R7
Ri	may be either R0 or R1; pointer registers
direct	may be any of the lower 128 internal RAM locations or one of the special function registers
B.ACC	is the 16 bit register consisting of register B, representing the high byte, and the accumulator, the low byte
XRAM[ ]	is a location in the external data space
CODE[ ]	denotes the external code memory

data	is an 8 bit constant
data16	is a 16 bit constant
bit	is either one of the 128 software flags from 0 to 127, located in the internal RAM at addresses 020H to 02FH, or one of the bits of a bit addressable special function register, numbered from 128 to 255
/bit	denotes the complement of a bit
addr11	is an 11 bit address; the upper 5 bits remain unchanged so the program counter will remain in the same 2 kB memory segment
addr16	denotes a 16 bit address
rel	is an address from -128 to +127 relative to the first code byte of the next instruction
[ ]16	denotes a 16 bit value
L[ ]	denotes the less significant nibble (4 bits) of a byte, and
H[ ]	the higher nibble (4 bits).
INT	is the integer value, and
MOD	the modulo or remainder of a division
AND, OR, XOR, NOT	are the basic Boolean operations
ROTATE	rotates the accumulator in the desired direction



ROTATE CY	rotates the accumulator through the carry



The encoding of the instructions can be found in [12]. It also provides timing diagrams of memory accesses.

Data Transfer

instruction	*operation*	*bytes*	*cycles*
MOV C,bit	[CY] := [bit]	2	1
MOV bit,C	[bit] := [CY]	2	2
MOV DPTR,#data16	[DPTR]16 := data	3	2
MOV A,#data	[ACC] := data	2	1
MOV A,direct	[ACC] := [direct]	2	1
MOV A,Rn	[ACC] := [Rn]	1	1
MOV A,@Ri	[ACC] := [[Ri]]	1	1
MOV Rn,#data	[Rn] := data	2	1
MOV Rn,direct	[Rn] := [direct]	2	2
MOV Rn,A	[Rn] := [ACC]	1	1
MOV direct,#data	[direct] := data	3	2
MOV direct,direct	[direct] := [direct]	3	2
MOV direct,Rn	[direct] := [Rn]	2	2
MOV direct,A	[direct] := [ACC]	2	1
MOV direct,@Ri	[direct] := [[Ri]]	2	2
MOV @Ri,#data	[[Ri]] := data	2	1
MOV @Ri,A	[[Ri]] := [ACC]	1	1
MOV @Ri,direct	[[Ri]] := [direct]	2	2
MOVC A,@A+DPTR	[ACC] := CODE[[A] + [DPTR]]	1	2
MOVC A,@A+PC	[ACC] := CODE[[A] + [PC]]	1	2
MOVX A,@Ri	[ACC] := XRAM[[Ri]]	1	2
	(address high byte from P2 SFR)		
MOVX A,@DPTR	[ACC] := XRAM[[DPTR]]	1	2
MOVX @Ri,A	XRAM[[Ri]] := [ACC]	1	2
	(address high byte from P2 SFR)		
MOVX @DPTR,A	XRAM[[DPTR]] := [ACC]	1	2
PUSH direct	[STACK] := [direct]	2	2
POP direct	[direct] := [STACK]	2	2
XCH A,direct	[ACC] := [direct]; [direct] := [ACC]	2	1
XCH A,Rn	[ACC] := [Rn]; [Rn] := [ACC]	1	1
XCH A,@Ri	[ACC] := [[Ri]; [[Ri]] := [ACC]	1	1
XCHD A,@Ri	L[ACC] := L[[Ri]]; L[[Ri]] := L[ACC]	1	1

Arithmetic Operations

instruction	*operation*	*bytes*	*cycles*
ADD A,#data	[ACC] := [ACC] + data	2	1
ADD A,direct	[ACC] := [ACC] + [direct]	2	1
ADD A,Rn	[ACC] := [ACC] + [Rn]	1	1
ADD A,@Ri	[ACC] := [ACC] + [[Ri]]	1	1
ADDC A,#data	[ACC] := [ACC] + [CY] + data	2	1
ADDC A,direct	[ACC] := [ACC] + [CY] + [direct]	2	1
ADDC A,Rn	[ACC] := [ACC] + [CY] + [Rn]	1	1
ADDC A,@Ri	[ACC] := [ACC] + [CY] + [[Ri]]	1	1
SUBB A,#data	[ACC] := [ACC] - [CY] - data	2	1
SUBB A,direct	[ACC] := [ACC] - [CY] - [direct]	2	1
SUBB A,Rn	[ACC] := [ACC] - [CY] - [Rn]	1	1
SUBB A,@Ri	[ACC] := [ACC] - [CY] - [[Ri]]	1	1
INC DPTR	[DPTR]16 := [DPTR]16 + 1	1	2
INC A	[ACC] := [ACC] + 1	1	1
INC direct	[direct] := [direct] + 1	2	1
INC Rn	[Rn] := [Rn] + 1	1	1
INC @Ri	[[Ri]] := [[Ri]] + 1	1	1
DEC A	[ACC] := [ACC] - 1	1	1
DEC direct	[direct] := [direct] - 1	2	1
DEC Rn	[Rn] := [Rn] - 1	1	1
DEC @Ri	[[Ri]] := [[Ri]] - 1	1	1
MUL AB	[B.ACC] := [ACC] * [B]	1	4
DIV AB	[ACC] := INT([ACC] / [B]); [B] := [ACC] MOD [B]	1	4
DA A	[ACC] decimal adjusted	1	1

Logical Operations

instruction	*operation*	*bytes*	*cycles*
ANL C,bit	[CY] := [CY] AND [bit]	2	2
ANL C,/bit	[CY] := [CY] AND NOT [bit]	2	2
ANL A,#data	[ACC] := [ACC] AND data	2	1
ANL A,direct	[ACC] := [ACC] AND [direct]	2	1
ANL A,Rn	[ACC] := [ACC] AND [Rn]	1	1
ANL A,@Ri	[ACC] := [ACC] AND [[Ri]]	1	1
ANL direct,A	[direct] := [direct] AND [ACC]	2	1
ANL direct,#data	[direct] := [direct] AND data	3	2
ORL C,bit	[CY] := [CY] OR [bit]	2	2
ORL C,/bit	[CY] := [CY] OR NOT [bit]	2	2
ORL A,#data	[ACC] := [ACC] OR data	2	1
ORL A,direct	[ACC] := [ACC] OR [direct]	2	1
ORL A,Rn	[ACC] := [ACC] OR [Rn]	1	1
ORL A,@Ri	[ACC] := [ACC] OR [[Ri]]	1	1
ORL direct,A	[direct] := [direct] OR [ACC]	2	1
ORL direct,#data	[direct] := [direct] OR data	3	2
XRL A,#data	[ACC] := [ACC] XOR data	2	1
XRL A,direct	[ACC] := [ACC] XOR [direct]	2	1
XRL A,Rn	[ACC] := [ACC] XOR [Rn]	1	1
XRL A,@Ri	[ACC] := [ACC] XOR [[Ri]]	1	1
XRL direct,A	[direct] := [direct] XOR [ACC]	2	1
XRL direct,#data	[direct] := [direct] XOR data	3	2
CLR C	[CY] := 0	1	1
CLR bit	[bit] := 0	2	1
CLR A	[ACC] := 00H	1	1
SETB C	[CY] := 1	1	1
SETB bit	[bit] := 1	2	1
CPL C	[CY] := NOT [CY]	1	1
CPL bit	[bit] := NOT [bit]	2	1
CPL A	[ACC] := NOT [ACC]	1	1
RL A	[ACC] := ROTATE LEFT [ACC]	1	1
RR A	[ACC] := ROTATE RIGHT [ACC]	1	1
RLC A	[ACC] := ROTATE LEFT CY [ACC]	1	1
RRC A	[ACC] := ROTATE RIGHT CY [ACC]	1	1
SWAP A	L[ACC] := H[ACC]; H[ACC] := L[ACC]	1	1

Program Branching

instruction	*operation*	*bytes*	*cycles*
ACALL addr11	subroutine call in 2 kB segment	2	2
LCALL addr16	subroutine call in 64 kB area	3	2
RET	return from subroutine	1	2
RETI	return from interrupt	1	2
AJMP addr11	absolute jump in 2 kB segment	2	2
LJMP addr16	absolute jump in 64 kB area	3	2
SJMP rel	jump relative to PC	2	2
JMP @A+DPTR	absolute jump to [ACC] + [DPTR]	1	2
JZ rel	relative jump if [ACC] = 00H	2	2
JNZ rel	relative jump if [ACC] <> 00H	2	2
JC rel	relative jump if [CY] = 1	2	2
JNC rel	relative jump if [CY] = 0	2	2
JB bit,rel	relative jump if [bit] = 1	3	2
JNB bit,rel	relative jump if [bit] = 0	3	2
JBC bit,rel	relative jump if [bit] = 1; [bit] := 0	3	2
CJNE A,#data,rel	relative jump if [ACC] <> data	3	2
CJNE A,direct,rel	relative jump if [ACC] <> [direct]	3	2
CJNE Rn,#data,rel	relative jump if [Rn] <> data	3	2
CJNE @Ri,#data,rel	relative jump if [[Ri]] <> data	3	2
DJNZ direct,rel	[direct] := [direct] - 1; relative jump if [direct] <> 00H	3	2
DJNZ Rn,rel	[Rn] := [Rn] - 1; relative jump if [Rn] <> 00H	2	2
NOP	no operation	1	1
res	illegal opcode (A5h)	.	.

Appendix D: File Formats

This Appendix describes the file formats of μOberon, namely the object and the symbol file produced by the compiler, and the Intel-Hex format that is used by the linker and interpreted by the decoder if the corresponding option is set. Alternatively, the linker produces a sequential binary file. Finally the structure of the options file is listed.

D.1 μOberon Object File Format

The object file with extension *obj* is generated by the compiler and contains the object code, i.e. the code bytes, the keys of the module and of the imported modules, and some fixup information for the linker. Note that the code block in this file will change considerably during the linking process, so it is not possible to use parts of it without linking. Object files may be decoded using the integrated μOberon decoder.

```
ObjectFile            = HeaderBlock ImportBlock EntryBlock CommandBlock
                        TrapProcedure InterruptBlock ConstantsBlock CodeBlock
                        ImportAccesses GlobalVarAccesses LocalVarAccesses
                        AddressAccesses TrapCalls.

HeaderBlock           = 0F1X nofImports:2 nofEntries:2 nofCommands:2
                        constantsSize:2 globalData:2 maxLocalData:2 codeSize:2
                        key:4 moduleName OptionsBlock.

OptionsBlock          = localDataWindow:2 intRAM:2 extRAMbeg:2 extRAMend:2.

ImportBlock           = { name key:4 }.
EntryBlock            = { entryAdr:2 }.
CommandBlock          = { name entryAdr:2 }.
TrapProcedure         = [ 0:2 entryAdr:2 ].
InterruptBlock        = { intAdr:2 entryAdr:2 } MarkTag:2.
ConstantsBlock        = { byte }.
CodeBlock             = { byte }.
ImportAccesses        = { ImportedProcCalls ImportedVarAccesses }.
ImportedProcCalls     = { entryNo:2 anchor:2 } MarkTag:2.
ImportedVarAccesses   = { pc:2 } MarkTag:2.
GlobalVarAccesses     = { pc:2 } MarkTag:2.
LocalVarAccesses      = { pc:2 } MarkTag:2.
AddressAccesses       = { pc:2 } MarkTag:2.
TrapCalls             = { pc:2 } MarkTag:2.
MarkTag               = 0FFFFH.
```

- Names are character sequences terminated by 0X. All other identifiers written with small letters stand for numbers.
- If a number is represented by more than one byte, the number of bytes is trailing the identifier after a colon.

D.2 µOberon Symbol File Format

The compiler also produces the symbol file with extension *sym*. The symbol file contains interface information that is used by clients importing that module. To decode a symbol file, use the integrated µOberon browser.

The constants written in capital letters may be found in the definition of the symbol table module XOT.

```
SymbolFile        = 0F3X MOD key:4 name {Element}.

Element           = MOD key:4 name
                    | CONST Constant
                    | (TYPE | HIDDENTYPE) ref modno name
                    | (VAR | RDONLYVAR) ref varno name
                    | (FLD | RDONLYFLD) ref offset:4 name
                    | (VALPAR | VARPAR) ref name
                    | PARLIST {Element} PROC ref procno name
                    | PTR baseRef modno
                    | PARLIST {Element} PROCTYPE resultRef modno
                    | ARRAY elemRef modno size:4
                    | FLDLIST {Element} REC baseRef modno size:4 descno:2
                    | (HIDDENPTR | HIDDENPROC) offset:4
                    | FIXUP ptrRef baseRef.

Constant          = (BYTE | CHAR | SINT) value name
                    | BOOL (FALSE | TRUE) name
                    | INT value:2 name
                    | (REAL | LINT | SET) value:4 name
                    | LREAL value:8 name
                    | STRING string name
                    | NIL name.
```

D.3 Intel-Hex File Format

The Intel-Hex format is a plain ASCII file with extension *hex* that contains the same information as the simple binary form plus some additional features like a checksum and the possibility to use different addresses not necessarily neighbouring. It only consists of the ASCII characters 0..9, A..F, the colon (:), as well as a carriage return (CR) and line feed (LF) at the end of each line. There are no blanks in the file.

The whole code is divided into blocks, and each line represents a block consisting of a defined number of code bytes. Consider the following short program:

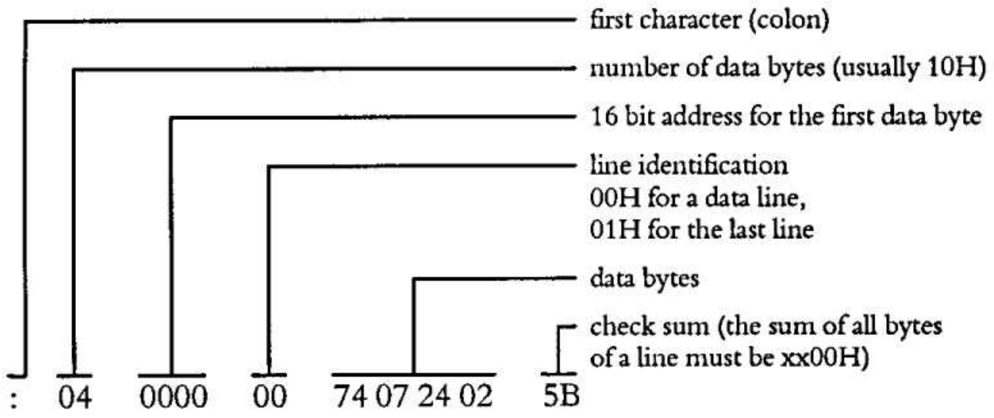
```
address   code    mnemonic

0000      74 07   MOV A,#07
0002      24 02   ADD A,#02
```

The corresponding Intel-Hex file would look as follows:

```
:0400000074072402 5B
:00000001FF
```

Each line consists of the following numbers, all written in plain ASCII hexadecimal format:


first character (colon)
number of data bytes (usually 10H)
16 bit address for the first data byte
line identification
00H for a data line,
01H for the last line
data bytes
check sum (the sum of all bytes of a line must be xx00H)
: 04 0000 00 74 07 24 02 5B

It is possible to decode an Intel-Hex file as well as a binary file using the µOberon core decoder, depending on the setting in the options panel.

D.4 µOberon Options File Format

In the *Rsrc* subdirectory there is a file called *Options* that contains the configuration data for the µOberon development system. This file has the following format:

```
OptionsFile          = NewSymFile LinkParam StackCheck
                       IndexCheck NilCheck ReturnCheck
                       LocalDataWindow:2 IntelHex:1 LptPort
                       IntRam:2 ExtRAMbeg:4 ExtRAMend:4.
```

- LinkParam and LptPort are character sequences terminated by 0X.
- Boolean values are written as a single byte of value 1 if TRUE or 0 respectively.

Appendix E: Data Structures and Module Interfaces

Refer to the diagram of the module hierarchy in chapter *Project µOberon*, section *Module Overview* for details.

E.1 Data Structures

Items

Items are entities that reflect factors, terms, variables and so on, that is, constituents of expressions and statements, and are generated while parsing ([1], [10]). Their type as well as the constants for the fields are defined in module XOT.

mode	*a0*	*a1*	*a2*	*description*
Undef				
Var	address			variable
RegI	0 or 1			register indirect
Ind	address			pointer indirect
ExtVar	address			variable in ext. RAM
Coc	relation	false jump	true jump	condition code
Stk				stack
Const	value	(value)		constant values
	address	length		strings
Reg	0 to 7			register
Fld	offset			record field
Typ				type
Proc	address	fixup chain		internal procedure
ImpProc	address	fixup chain		imported procedure
StdProc	function nr.			standard procedure
IntProc	address	fixup chain		interrupt procedure
TrapProc	address	fixup chain		trap procedure
Mod	module nr.	module key		module
Head	level of scope			

E.2 Module Interfaces

The following module interfaces are simplified.

Host Interface XOH

```
DEFINITION Mcs51XOH;

    CONST
        ObjFileTag = F1X;
        ObjectFileExtension = "obj";
        SymFileTag = F3X;
        SymbolFileExtension = "sym";
        BinaryFileExtension = "bin";
        IntelHexFileExtension = "hex";
    MarkTag = 65535;
    MaxCodeSize = 65536;
    MaxIdLen = 32;
    MaxNofImports = 32;
    MaxStrLen = 256;

TYPE
    InputFile = POINTER TO RECORD
            name-: Files.Name;
            PROCEDURE (in: InputFile) Close;
            PROCEDURE (in: InputFile) Eof (): BOOLEAN;
            PROCEDURE (in: InputFile) Pos (): LONGINT;
            PROCEDURE (in: InputFile) Read (VAR ch: CHAR);
            PROCEDURE (in: InputFile) ReadBool (VAR b: BOOLEAN);
            PROCEDURE (in: InputFile) ReadBytes (VAR bytes: ARRAY OF CHAR; nofBytes: LONGINT);
            PROCEDURE (in: InputFile) ReadHex8 (VAR i: LONGINT);
            PROCEDURE (in: InputFile) ReadInt (VAR i: INTEGER);
            PROCEDURE (in: InputFile) ReadLInt (VAR l: LONGINT);
            PROCEDURE (in: InputFile) ReadSInt (VAR s: SHORTINT);
            PROCEDURE (in: InputFile) ReadString (VAR str: ARRAY OF CHAR);
            PROCEDURE (in: InputFile) ReadWord (VAR l: LONGINT);
            PROCEDURE (in: InputFile) SetPos (pos: LONGINT)
        END;

    OutputFile = POINTER TO RECORD
            name-: Files.Name;
            PROCEDURE (out: OutputFile) Close;
            PROCEDURE (out: OutputFile) Pos (): LONGINT;
            PROCEDURE (out: OutputFile) Register;
            PROCEDURE (out: OutputFile) SetPos (pos: LONGINT);
            PROCEDURE (out: OutputFile) Write (ch: CHAR);
            PROCEDURE (out: OutputFile) WriteBool (b: BOOLEAN);
            PROCEDURE (out: OutputFile) WriteBytes (VAR bytes: ARRAY OF CHAR; nofBytes: LONGINT);
            PROCEDURE (out: OutputFile) WriteHex8 (i: LONGINT);
            PROCEDURE (out: OutputFile) WriteInt (i: INTEGER);
            PROCEDURE (out: OutputFile) WriteLInt (l: LONGINT);
            PROCEDURE (out: OutputFile) WriteSInt (s: SHORTINT);
            PROCEDURE (out: OutputFile) WriteString (str: ARRAY OF CHAR);
            PROCEDURE (out: OutputFile) WriteWord (l: LONGINT)
        END;

    Window = POINTER TO RECORD
            PROCEDURE (win: Window) Open (title: ARRAY OF CHAR);
            PROCEDURE (win: Window) SetRuler (tabs: ARRAY OF LONGINT);
            PROCEDURE (win: Window) Write (ch: CHAR);
            PROCEDURE (win: Window) WriteHex16 (l: LONGINT);
            PROCEDURE (win: Window) WriteHex32 (l: LONGINT);
            PROCEDURE (win: Window) WriteHex8 (i: LONGINT);
            PROCEDURE (win: Window) WriteInt (i: LONGINT);
            PROCEDURE (win: Window) WriteLn;
            PROCEDURE (win: Window) WriteString (str: ARRAY OF CHAR);
            PROCEDURE (win: Window) WriteTab
        END;

    Id = ARRAY 32 OF CHAR;

    Str = ARRAY 256 OF CHAR;

VAR
    path: ARRAY 32 OF CHAR;

PROCEDURE CloseDialog;
PROCEDURE GetClock (VAR time, date: LONGINT);
PROCEDURE [code] InitLPT (port: INTEGER): BOOLEAN;
PROCEDURE InputFileFrom (out: OutputFile): InputFile;
PROCEDURE InsertError (nr: INTEGER);
PROCEDURE LogWrite (ch: CHAR);
PROCEDURE LogWriteHex16 (l: LONGINT);
PROCEDURE LogWriteHex32 (l: LONGINT);
PROCEDURE LogWriteHex8 (i: LONGINT);
PROCEDURE LogWriteInt (i: LONGINT);
PROCEDURE LogWriteLn;
PROCEDURE LogWriteString (s: ARRAY OF CHAR);
PROCEDURE NewInputFile (path, name, type: ARRAY OF CHAR): InputFile;
PROCEDURE NewOutputFile (path, name, type: ARRAY OF CHAR): OutputFile;
PROCEDURE NewWindow (): Window;
PROCEDURE OpenToolDialog (fileName, dialogTitle: ARRAY OF CHAR);
PROCEDURE [code] OutLPT (port: INTEGER; ch: CHAR);
```

```
    PROCEDURE SelectInputFile (path, defaultType: ARRAY OF CHAR): InputFile;
    PROCEDURE ShowFirstError;
    PROCEDURE SourceEot (): BOOLEAN;
    PROCEDURE SourceOpen (): BOOLEAN;
    PROCEDURE SourcePos (): LONGINT;
    PROCEDURE SourceRead (VAR ch: CHAR);

END Mcs51XOH.
```

Scanner XOS

```
DEFINITION Mcs51XOS;

    VAR
        id-: Mcs51XOH.Id;
        str-: Mcs51XOH.Str;
        numTyp-: SHORTINT;
        intVal-: LONGINT;
        realVal-: REAL;
        lrealVal-: LONGREAL;
        scanErr-: LONGINT;

    PROCEDURE Get (VAR sym: SHORTINT);
    PROCEDURE Init;
    PROCEDURE Mark (nr: INTEGER);

END Mcs51XOS.
```

Symbol Table XOT

```
DEFINITION Mcs51XOT;

    TYPE
        Object = POINTER TO ObjDesc;
        ObjDesc = RECORD
            mode: SHORTINT;
            typ: Struct;
            a0, a1, a2: LONGINT;
            name: Mcs51XOH.Id;
            marked: BOOLEAN;
            dsc, next: Object
        END;

        Struct = POINTER TO StrDesc;
        StrDesc = RECORD
            form: SHORTINT;
            extLev, mno: INTEGER;
            ref: SHORTINT;
            size, adr: LONGINT;
            BaseTyp: Struct;
            link, structObj: Object
        END;

        Item = RECORD
            mode: SHORTINT;
            typ: Struct;
            a0, a1, a2: LONGINT;
            lev: INTEGER;
            obj: Object
        END;

    VAR
        topScope: Object;
        nofEntries-: INTEGER;
        entry-: ARRAY 96 OF INTEGER;
        importedModule-: ARRAY 32 OF Object;
        nofImports-: INTEGER;
        systemImported-: BOOLEAN;

    PROCEDURE Close;
    PROCEDURE CloseScope;
    PROCEDURE Export (VAR moduleName: Mcs51XOH.Id; VAR newSymFile: BOOLEAN; VAR key: LONGINT);
    PROCEDURE Find (VAR result: Object; VAR level: INTEGER);
    PROCEDURE FindField (typ: Struct; VAR result: Object);
    PROCEDURE FindImport (mod: Object; VAR result: Object);
    PROCEDURE Import (VAR importName, importAlias, self: Mcs51XOH.Id);
    PROCEDURE Init;
    PROCEDURE Insert (name: Mcs51XOH.Id; VAR result: Object);
    PROCEDURE OpenScope (level: INTEGER);

END Mcs51XOT.
```

Code Generator XOC

```
DEFINITION Mcs51XOC;

    VAR
        intRAM: INTEGER;
        extRAMbeg: LONGINT;
        extRAMend: LONGINT;
        localDataWindow: INTEGER;
        stackCheck: BOOLEAN;
        indexCheck: BOOLEAN;
        nilCheck: BOOLEAN;
        returnCheck: BOOLEAN;
        level: INTEGER;
        pc-: LONGINT;

    PROCEDURE AddressAccess (at: LONGINT);
    PROCEDURE AllocString (VAR x: Mcs51XOT.Item);
    PROCEDURE BJump (location: LONGINT);
    PROCEDURE CBJump (VAR x: Mcs51XOT.Item; cond: BOOLEAN; location: LONGINT);
    PROCEDURE CFJump16 (VAR x: Mcs51XOT.Item; cond: BOOLEAN; VAR location: LONGINT);
    PROCEDURE Call (VAR x: Mcs51XOT.Item; paramSize: LONGINT);
    PROCEDURE CompareParLists (x, y: Mcs51XOT.Object);
    PROCEDURE Enter (mode: SHORTINT; paramSize, dataSize: LONGINT);
    PROCEDURE FJump16 (VAR location: LONGINT);
    PROCEDURE FJump8 (VAR location: LONGINT);
    PROCEDURE Fix16 (at, with: LONGINT);
    PROCEDURE Fix8 (at, with: LONGINT);
    PROCEDURE FixupChain16 (location: LONGINT);
    PROCEDURE FixupChain8 (location: LONGINT);
    PROCEDURE FreeItem (VAR x: Mcs51XOT.Item);
    PROCEDURE GetPtrReg (VAR x: Mcs51XOT.Item);
    PROCEDURE GetReg (VAR x: Mcs51XOT.Item);
    PROCEDURE GetResReg (VAR x: Mcs51XOT.Item);
    PROCEDURE GlobalVarAccess (at: LONGINT);
    PROCEDURE ImportedVarAccess (at, moduleNr: LONGINT);
    PROCEDURE Init;
    PROCEDURE InsertBranch8 (VAR x: Mcs51XOT.Item; cond: BOOLEAN);
    PROCEDURE Inverted (relation: LONGINT): SHORTINT;
    PROCEDURE Leave (mode: SHORTINT; dataSize: LONGINT; fctProc: BOOLEAN);
    PROCEDURE LoadA (x: Mcs51XOT.Item; byteNr: LONGINT);
    PROCEDURE LoadAdrA (VAR x: Mcs51XOT.Item);
    PROCEDURE LoadAdrB (VAR x: Mcs51XOT.Item);
    PROCEDURE LoadAdrR (rn: LONGINT; x: Mcs51XOT.Item);
    PROCEDURE LoadB (x: Mcs51XOT.Item; byteNr: LONGINT);
    PROCEDURE LoadCaseExpression (x: Mcs51XOT.Item);
    PROCEDURE LoadR (rn: LONGINT; x: Mcs51XOT.Item; byteNr: LONGINT);
    PROCEDURE LocalVarAccess (at: LONGINT);
    PROCEDURE MergedLinks8 (L0, L1: LONGINT): LONGINT;
    PROCEDURE MoveBlock (VAR x, y: Mcs51XOT.Item; nofBytes: LONGINT);
    PROCEDURE OutCode (VAR moduleName: Mcs51XOH.Id; key, globalData, maxLocalData: LONGINT);
    PROCEDURE ParameterList (VAR x: Mcs51XOT.Item): Mcs51XOT.Object;
    PROCEDURE Pop (VAR x: Mcs51XOT.Item);
    PROCEDURE PopRegisters (toRestore: SET);
    PROCEDURE Push (VAR x: Mcs51XOT.Item);
    PROCEDURE PushAdr (VAR x: Mcs51XOT.Item);
    PROCEDURE PushUsedRegisters (VAR savedRegs: SET);
    PROCEDURE PutByte (byte: LONGINT);
    PROCEDURE PutF1 (instr: LONGINT);
    PROCEDURE PutF2 (instr, op: LONGINT);
    PROCEDURE PutF3 (instr, op1, op2: LONGINT);
    PROCEDURE SetCC (VAR x: Mcs51XOT.Item; relation: SHORTINT);
    PROCEDURE SetIntType (VAR x: Mcs51XOT.Item);
    PROCEDURE Store (VAR x, y: Mcs51XOT.Item);
    PROCEDURE StoreA (VAR x: Mcs51XOT.Item; byteNr: LONGINT);
    PROCEDURE StoreB (VAR x: Mcs51XOT.Item; byteNr: LONGINT);
    PROCEDURE Trap (n: LONGINT);
    PROCEDURE VarAccess (at: LONGINT; level: INTEGER);

END Mcs51XOC.
```

Arithmetic Engine XOA

```
DEFINITION Mcs51XOA;

    PROCEDURE Abs (VAR x: Mcs51XOT.Item);
    PROCEDURE Compare (op: SHORTINT; VAR x, y: Mcs51XOT.Item);
    PROCEDURE Dec (VAR x: Mcs51XOT.Item);
    PROCEDURE Div (VAR x, y: Mcs51XOT.Item);
    PROCEDURE Entier (VAR x: Mcs51XOT.Item);
    PROCEDURE Inc (VAR x: Mcs51XOT.Item);
    PROCEDURE Long (VAR x: Mcs51XOT.Item; typ: Mcs51XOT.Struct);
    PROCEDURE Minus (VAR x, y: Mcs51XOT.Item);
    PROCEDURE Mod (VAR x, y: Mcs51XOT.Item);
    PROCEDURE Negate (VAR x: Mcs51XOT.Item);
    PROCEDURE Plus (VAR x, y: Mcs51XOT.Item);
    PROCEDURE Short (VAR x: Mcs51XOT.Item; typ: Mcs51XOT.Struct);
    PROCEDURE Slash (VAR x, y: Mcs51XOT.Item);
    PROCEDURE Times (VAR x, y: Mcs51XOT.Item);

END Mcs51XOA.
```

Expression Handler XOE

```
DEFINITION Mcs51XOE;

    CONST
        MaxLoopNesting = 16;
        MaxNofCases = 128;

    PROCEDURE Assign (VAR x, y: Mcs51XOT.Item);
    PROCEDURE CaseCmpRange (eform: SHORTINT; from, to: LONGINT);
    PROCEDURE CaseCmpValue (eform: SHORTINT; val: LONGINT);
    PROCEDURE DeReference (VAR x: Mcs51XOT.Item);
    PROCEDURE Field (VAR rec: Mcs51XOT.Item; fld: Mcs51XOT.Object);
    PROCEDURE In (VAR x, y: Mcs51XOT.Item);
    PROCEDURE Index (VAR x, y: Mcs51XOT.Item);
    PROCEDURE Op1 (op: INTEGER; VAR x: Mcs51XOT.Item);
    PROCEDURE Op2 (op: SHORTINT; VAR x, y: Mcs51XOT.Item);
    PROCEDURE PushParameter (VAR par: Mcs51XOT.Item; type: Mcs51XOT.Object);
    PROCEDURE Result (VAR x: Mcs51XOT.Item; typ: Mcs51XOT.Struct);
    PROCEDURE Set0 (VAR x, y: Mcs51XOT.Item);
    PROCEDURE Set1 (VAR x, y, z: Mcs51XOT.Item);
    PROCEDURE StdProcedure (fctNo, nofPar: SHORTINT; VAR x, y, z: Mcs51XOT.Item);

END Mcs51XOE.
```

Inline Assembler XOI

```
DEFINITION Mcs51XOI;

    PROCEDURE Assembler (VAR parserSym: SHORTINT);

END Mcs51XOI.
```

Parser XOP

```
DEFINITION Mcs51XOP;

    VAR
        newSymFile: BOOLEAN;

    PROCEDURE Compile;

END Mcs51XOP.
```

Linker and Loader XOL

```
DEFINITION Mcs51XOL;

    VAR
        param: Mcs51XOH.Str;
        intelHex: BOOLEAN;
        lptPort: INTEGER;
    PROCEDURE Link;
    PROCEDURE Load;

END Mcs51XOL.
```

Browser XOB

```
DEFINITION Mcs51XOB;

    PROCEDURE ShowDef;

END Mcs51XOB.
```

Decoder XOD

```
DEFINITION Mcs51XOD;

    VAR
        intelHex: BOOLEAN;

    PROCEDURE DecodeCore;
    PROCEDURE DecodeObj;

END Mcs51XOD.
```

User Interface MicroOberon

```
DEFINITION Mcs51MicroOberon;

    VAR
        options: RECORD
            newSymFile: BOOLEAN;
            linkParam: Mcs51XOH.Str;
            stackCheck, indexCheck, nilCheck, returnCheck: BOOLEAN;
            localDataWindow: INTEGER;
            intelHex: SHORTINT;
            lptPort: LptPort;
            intRAM: INTEGER;
            extRAMbeg, extRAMend: LONGINT
        END;

    PROCEDURE Browse;
    PROCEDURE Cancel;
    PROCEDURE Compile;
    PROCEDURE DecodeLink;
    PROCEDURE DecodeObj;
    PROCEDURE Info;
    PROCEDURE Link;
    PROCEDURE Load;
    PROCEDURE LoadOptions;
    PROCEDURE Ok;
    PROCEDURE OpenMainTool;
    PROCEDURE OpenOptionsTool;
    PROCEDURE SaveOptions;

END Mcs51MicroOberon.
```

Appendix F: Bibliography

About Oberon

[1] N. Wirth, J. Gutknecht, *Project Oberon, The Design of an Operating System and Compiler.* Addison-Wesley, 1993.
[2] M. Reiser, N. Wirth, *Programming in Oberon, Steps beyond Pascal and Modula.* Addison-Wesley, 1992.
[3] N. Wirth, *The Programming Language Oberon.* Institute for Computer Systems, ETH Zürich, 1990.
[4] H. Mössenböck, N. Wirth, *The Programming Language Oberon-2.* Institute for Computer Systems, ETH Zürich, 1992.
[5] M. Brandis, R. Crelier, M. Franz, J. Templ, *The Oberon System Family.* Institute for Computer Systems, ETH Zürich.
[6] C. Pfister, B. Heeb, J. Templ, *Oberon Technical Notes, Report 156.* Institute for Computer Systems, ETH Zürich, 1991.
[7] N. Wirth, *From Modula to Oberon.* Institute for Computer Systems, ETH Zürich, 1990.
[8] N. Wirth, *Tasks versus Threads, An Alternative Multiprocessing Paradigm.* Software-Concepts and Tools, no. 17, 1996, pp. 6-12.

Compiler Construction

[9] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers, Principles, Techniques and Tools.* Addison-Wesley, 1986.
[10] N. Wirth, *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, 1996.
[11] G. Rivera, I. Salvetti, *Oberon-Crosscompiler, Monitor und Kernel für 68HC11.* Diploma theses, Institute for Computer Systems, ETH Zürich, 1996.
also see [1]

Microcontrollers

[12] A. Roth, *Das Mikrocontroller - Kochbuch.* IWT Verlag GmbH, 1990.
[13] A. Roth, *Das Mikrocontroller - Applikations - Kochbuch.* IWT Verlag GmbH, 1996.
[14] M. Ohsmann, *Mikrocontroller Handbuch.* Elektor-Verlag, 1993.
[15] O'Niel V. Som, *Pascal - Cross-Compiler für Mikrocontroller der 8031-Familie.* Elektor-Verlag, 1995.
[16] E. Esders, *Singlechip-Prozessoren, On-Chip-Peripherie und Kommunikationsmöglichkeiten moderner Mikrocontroller.* Franzis-Verlag, 1987.
[17] *16-bit 80C51XA Microcontrollers (eXtended Architecture).* Data Handbook IC25, Philips Semiconductors, 1995.
[18] *8051 microcontroller frequently asked questions list.* comp.sys.intel, last modified Feb 26th 1994.

Appendix G: Index