Section 4

AVR Assembler User Guide

4.1 Introduction

Welcome to the Atmel *AVR* Assembler. This manual describes the usage of the Assembler. The Assembler covers the whole range of microcontrollers in the AT90S family.

The Assembler translates assembly source code into object code. The generated object code can be used as input to a simulator or an emulator such as the Atmel *AVR* In-Circuit Emulator. The Assembler also generates a PROMable code and an optional EEPROM file which can be programmed directly into the program memory and EEPROM memory of an *AVR* microcontroller.

The Assembler generates fixed code allocations, consequently no linking is necessary.

The Assembler runs under Microsoft Windows 3.11, Microsoft Windows95 and Microsoft Windows NT. In addition, there is an MS-DOS command line version. The Windows version of the program contains an on-line help function covering most of this document.

The instruction set of the *AVR* family of microcontrollers is only briefly described, refer to the *AVR* Data Book (also available on CD-ROM) in order to get more detailed knowledge of the instruction set for the different microcontrollers.

To get quickly started, the Quick-Start Tutorial is an easy way to get familiar with the Atmel *AVR* Assembler.

4.2 Assembler Quick Start Tutorial

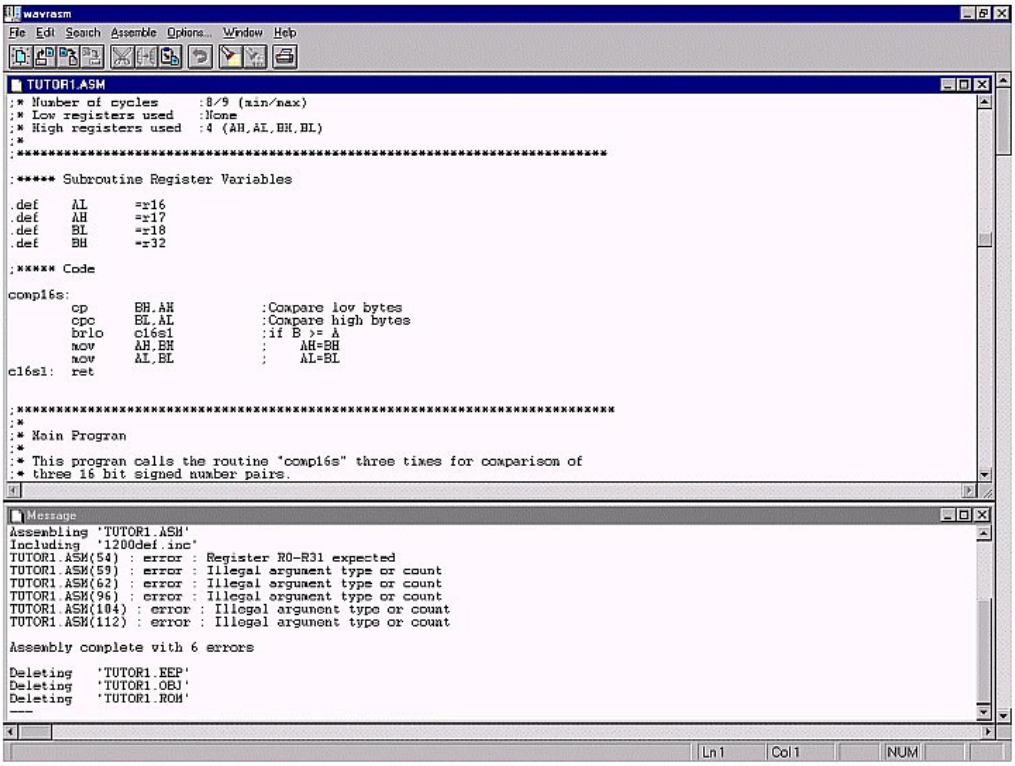
This tutorial assumes that the *AVR* Assembler and all program files that come with it are properly installed on your computer. Please refer to the installation instructions

4.2.1 Getting Started

Start the *AVR* Assembler. By selecting “File → Open” from the menu or by clicking on the toolbar, open the file “tutor1.asm”. This loads the assembly file into the Editor window. Read the program header and take a look at the program but do not make any changes yet.

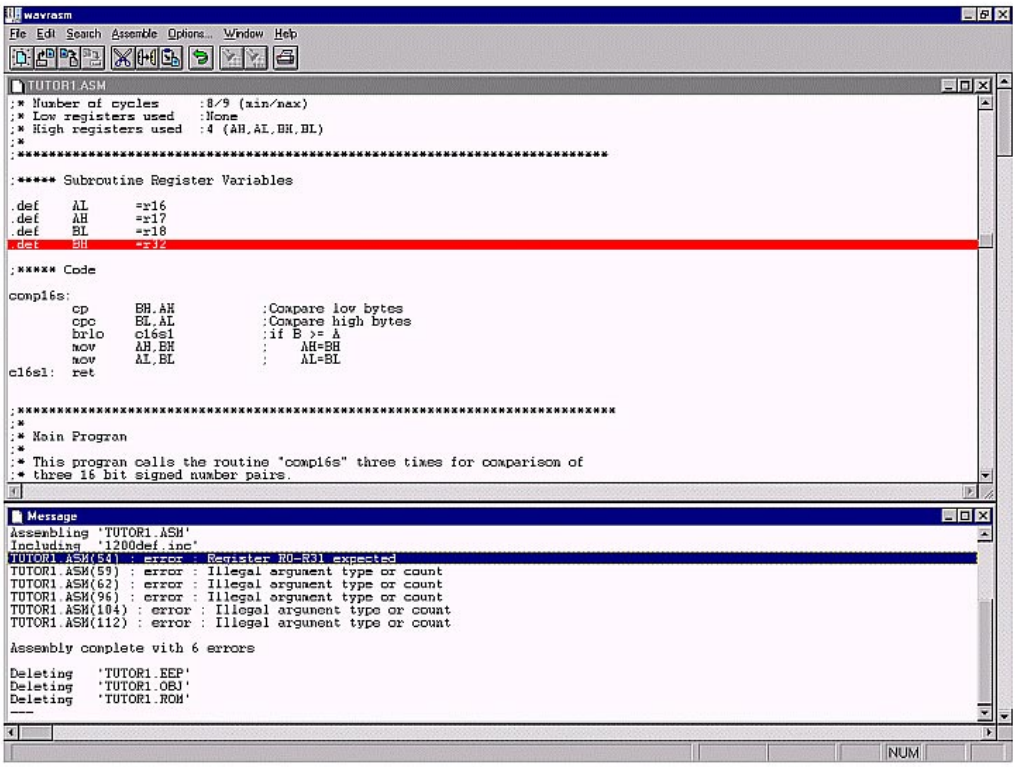
4.2.2 Assembling Your First File

Once you have had a look at the program, select Assemble from the menu. A second window (the Message window) appears, containing a lot of error messages. This window will overlap the editor window, so it is a good idea to clean up your work space on the screen. Select the Editor window containing the program code, and select “Window → Tile Horizontal” from the menu. It is useful to have the Editor window larger than the Message window, so move the top of the Message window down a bit, and follow with the bottom of the Editor window. Your screen should look like this:



4.2.3 Finding and Correcting Errors

From the looks of the Message window, it seems that you have attempted to assemble a program with lots of bugs. To get any further, the errors must be found and corrected. Point to the first error message in the Message window (the one reported to be on line 54) and press the left mouse button. Notice that in the Editor window, a red vertical bar is displayed all over line 54. The error message says that only registers R0 to R31 can be assigned variable names. That is true since the *AVR* has exactly 32 General Purpose working registers numbered R0-R31, and “tutor1.asm” tries to assign a name to register 32. See the figure below.



Double click on the error message in the Message window and observe that the Editor window becomes active while the cursor is positioned at the start of the line containing the error. Correct the mistake by changing “r32” to “r19” in the Editor window. One down, five to go.

Now click on the next error in the list. The message “Illegal argument type or count”, tells that something is wrong with the arguments following the compare (“cp”) instruction. Notice that the register named “BH” is one of the arguments, which happens to be the variable we just corrected. By clicking along on the remaining errors, it appears that the first error generated all the messages.

4.2.4 Reassembling

To find out whether all errors have been corrected, double click on any error (to activate the Editor window) or click inside the Editor window before you assemble once more. If you have done it all right up till now, the Message window will tell that you are crowned with success.

4.3 Assembler source

The Assembler works on source files containing instruction mnemonics, labels and directives. The instruction mnemonics and the directives often take operands.

Code lines should be limited to 120 characters.

Every input line can be preceded by a label, which is an alphanumeric string terminated by a colon. Labels are used as targets for jump and branch instructions and as variable names in Program memory and RAM.

An input line may take one of the four following forms:

1. **`[label:] directive [operands] [Comment]`**
2. **`[label:] instruction [operands] [Comment]`**
3. **`Comment`**
4. **`Empty line`**

A comment has the following form:

`; [Text]`

Items placed in braces are optional. The text between the comment-delimiter (;) and the end of line (EOL) is ignored by the Assembler. Labels, instructions and directives are described in more detail later.

Examples:

```
label:   .EQU var1=100     ; Set var1 to 100 (Directive)
         .EQU var2=200     ; Set var2 to 200
test:    rjmp  test      ; Infinite loop (Instruction)
                           ; Pure comment line
                           ; Another comment line
```

Note: *There are no restrictions with respect to column placement of labels, directives, comments or instructions.*

4.4 Instruction mnemonics

The Assembler accepts mnemonic instructions from the instruction set. A summary of the instruction set mnemonics and their parameters is given here. For a detailed description of the Instruction set, refer to the *AVR* Data Book.

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add without Carry	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rd, K	Add Immediate to Word	Rd+1:Rd ← Rd+1:Rd + K	Z,C,N,V	2
SUB	Rd, Rr	Subtract without Carry	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Immediate	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract Immediate with Carry	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rd, K	Subtract Immediate from Word	Rd+1:Rd ← Rd+1:Rd - K	Z,C,N,V	2
AND	Rd, Rr	Logical AND	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND with Immediate	Rd ← Rd • K	Z,N,V	1
OR	Rd, Rr	Logical OR	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR with Immediate	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← $FF - Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← $00 - Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • ($FFh - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd - 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← $FF	None	1
MUL	Rd,Rr	Multiply Unsigned	R1, R0 ← Rd × Rr	C	2 (1)

Note: 1. Not available in base-line microcontrollers

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
JMP	k	Jump	PC ← k	None	3
RCALL	k	Relative Call Subroutine	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
CALL	k	Call Subroutine	PC ← k	None	4
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None	1 / 2 / 3
CP	Rd,Rr	Compare	Rd - Rr	Z,C,N,V,H	1
CPC	Rd,Rr	Compare with Carry	Rd - Rr - C	Z,C,N,V,H	1
CPI	Rd,K	Compare with Immediate	Rd - K	Z,C,N,V,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBRS	Rr, b	Skip if Bit in Register Set	if (Rr(b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if(I/O(P,b)=0) PC ← PC + 2 or 3	None	1 / 2 / 3
SBIS	P, b	Skip if Bit in I/O Register Set	if(I/O(P,b)=1) PC ← PC + 2 or 3	None	1 / 2 / 3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC ← PC+k + 1	None	1 / 2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC ← PC+k + 1	None	1 / 2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1 / 2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1 / 2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1 / 2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1 / 2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC+ k + 1	None	1 / 2
BRLT	k	Branch if Less Than, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1 / 2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1 / 2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1 / 2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1 / 2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1 / 2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1 / 2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1 / 2
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1 / 2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1 / 2

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Copy Register	Rd ← Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	3
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Increment	Rd ← (X), X ← X + 1	None	2
LD	Rd, -X	Load Indirect and Pre-Decrement	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-Increment	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, -Y	Load Indirect and Pre-Decrement	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-Increment	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load Indirect and Pre-Decrement	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	3
ST	X, Rr	Store Indirect	(X) ← Rr	None	2
ST	X+, Rr	Store Indirect and Post-Increment	(X) ← Rr, X ← X + 1	None	2
ST	-X, Rr	Store Indirect and Pre-Decrement	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-Increment	(Y) ← Rr, Y ← Y + 1	None	2
ST	-Y, Rr	Store Indirect and Pre-Decrement	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-Increment	(Z)← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-Decrement	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q)← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2

Mnemonics	Operands	Description	Operation	Flags	#Clock Note
BIT AND BIT-TEST INSTRUCTIONS					
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n),Rd(0) ← 0,C ← Rd(7)	Z,C,N,V,H	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1),Rd(7) ← 0,C ← Rd(0)	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0) ← C,Rd(n+1) ← Rd(n),C ← Rd(7)	Z,C,N,V,H	1
ROR	Rd	Rotate Right Through Carry	Rd(7) ← C,Rd(n) ← Rd(n+1),C ← Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0) ↔ Rd(7..4)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
SBI	P, b	Set Bit in I/O Register	I/O(P, b) ← 1	None	2
CBI	P, b	Clear Bit in I/O Register	I/O(P, b) ← 0	None	2
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Two's Complement Overflow	V ← 1	V	1
CLV		Clear Two's Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ←1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1
NOP		No Operation		None	1
SLEEP		Sleep		None	1
WDR		Watchdog Reset		None	1

The Assembler is not case sensitive.

The operands have the following forms:

- Rd: R0-R31 or R16-R31 (depending on instruction)
- Rr: R0-R31
- b: Constant (0-7), can be a constant expression
- s: Constant (0-7), can be a constant expression
- P: Constant (0-31/63), can be a constant expression
- K: Constant (0-255), can be a constant expression
- k: Constant, value range depending on instruction. Can be a constant expression.
- q: Constant (0-63), can be a constant expression

4.5 Assembler directives

The Assembler supports a number of directives. The directives are not translated directly into opcodes. Instead, they are used to adjust the location of the program in memory, define macros, initialize memory and so on. An overview of the directives is given in the following table.

Summary of directives:

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define constant word(s)
ENDMACRO	End macro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn macro expansion on
MACRO	Begin macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

Note: *All directives must be preceded by a period.*

4.5.1 BYTE - Reserve bytes to a variable

The BYTE directive reserves memory resources in the SRAM. In order to be able to refer to the reserved location, the BYTE directive should be preceded by a label. The directive takes one parameter, which is the number of bytes to reserve. The directive can only be used within a Data Segment (see directives CSEG, DSEG and ESEG). Note that a parameter must be given. The allocated bytes are not initialized.

Syntax:

```
LABEL:  .BYTE expression
```

Example:

```
.DSEG
        var1:   .BYTE 1         ; reserve 1 byte to var1
        table: .BYTE tab_size   ; reserve tab_size bytes
.CSEG
        ldi     r30,low(var1)   ; Load Z register low
        ldi     r31,high(var1)  ; Load Z register high
        ld      r1,Z            ; Load VAR1 into register 1
```

4.5.2 CSEG - Code Segment

The CSEG directive defines the start of a Code Segment. An Assembler file can consist of several Code Segments, which are concatenated into one Code Segment when assembled. The BYTE directive can not be used within a Code Segment. The default segment type is Code. The Code Segments have their own location counter which is a word counter. The ORG directive (see description later in this document) can be used to place code and constants at specific locations in the Program memory. The directive does not take any parameters.

Syntax:

```
.CSEG
```

Example:

```
.DSEG                           ; Start data segment
        vartab: .BYTE 4         ; Reserve 4 bytes in SRAM
.CSEG                           ; Start code segment
        const:  .DW 2           ; Write 0x0002 in prog.mem.
        mov r1,r0               ; Do something
```

4.5.3 DB-Define constant byte(s) in program memory or E^{2}PROM memory

The DB directive reserves memory resources in the program memory or the EEPROM memory. In order to be able to refer to the reserved locations, the DB directive should be preceded by a label.

The DB directive takes a list of expressions, and must contain at least one expression. The DB directive must be placed in a Code Segment or an EEPROM Segment.

The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -128 and 255. If the expression evaluates to a negative number, the 8 bits two's complement of the number will be placed in the program memory or EEPROM memory location.

If the DB directive is used in a Code Segment and the expressionlist contains more than one expression, the expressions are packed so that two bytes are placed in each program memory word. If the expressionlist contains an odd number of expressions, the last expression will be placed in a program memory word of its own, even if the next line in the assembly code contains a DB directive.

Syntax:

```
LABEL: .DB expressionlist
```

Example:

```
.CSEG
        consts: .DB 0, 255, 0b01010101, -128, 0xaa
.ESEG
        eeconst:.DB 0xff
```

4.5.4 DEF - Set a symbolic name on a register

The DEF directive allows the registers to be referred to through symbols. A defined symbol can be used in the rest of the program to refer to the register it is assigned to. A register can have several symbolic names attached to it. A symbol can be redefined later in the program.

Syntax:

```
.DEF Symbol=Register
```

Example:

```
.DEF temp=R16
.DEF ior=R0
.CSEG
        ldi     temp,0xf0     ; Load 0xf0 into temp register
        in      ior,0x3f      ; Read SREG into ior register
        eor     temp,ior      ; Exclusive or temp and ior
```

4.5.5 DEVICE - Define which device to assemble for

The DEVICE directive allows the user to tell the Assembler which device the code is to be executed on. If this directive is used, a warning is issued if an instruction not supported by the specified device occurs in the code. If the size of the Code Segment or EEPROM Segment is larger than supported by the specified device, a warning is issued. If the DEVICE directive is not used, it is assumed that all instructions are supported and that there are no restrictions on memory sizes.

Syntax:

```
.DEVICE AT90S1200 | AT90S2313 | AT90S4414 | AT90S8515
```

Example:

```
.DEVICE AT90S1200             ; Use the AT90S1200
.CSEG
         push    r30          ; This statement will generate
                              ; a warning since the
                              ; specified device does not
                              ; have this instruction
```

4.5.6 DSEG - Data Segment

The DSEG directive defines the start of a Data Segment. An Assembler file can consist of several Data Segments, which are concatenated into one Data Segment when assembled. A Data Segment will normally only consist of BYTE directives (and labels). The Data Segments have their own location counter which is a byte counter. The ORG directive (see description later in this document) can be used to place the variables at specific locations in the SRAM. The directive does not take any parameters.

Syntax:

```
.DSEG
```

Example:

```
.DSEG                           ; Start data segment
       var1:.BYTE 1             ; reserve 1 byte to var1
       table:.BYTE tab_size     ; reserve tab_size bytes.
.CSEG
       ldi      r30,low(var1)   ; Load Z register low
       ldi      r31,high(var1)  ; Load Z register high
       ld       r1,Z            ; Load var1 into register 1
```

4.5.7 DW-Define constant word(s) in program memory or E^{2}PROM memory

The DW directive reserves memory resources in the program memory or EEPROM memory. In order to be able to refer to the reserved locations, the DW directive should be preceded by a label.

The DW directive takes a list of expressions, and must contain at least one expression.

The DB directive must be placed in a Code Segment or an EEPROM Segment.

The expression list is a sequence of expressions, delimited by commas. Each expression must evaluate to a number between -32768 and 65535. If the expression evaluates to a negative number, the 16 bits two's complement of the number will be placed in the program memory location.

Syntax:

```
LABEL: .DW expressionlist
```

Example:

```
.CSEG
        varlist:.DW 0,0xffff,0b1001110001010101,-32768,65535
.ESEG
        eevar:  .DW 0xffff
```

4.5.8 ENDMACRO - End macro

The ENDMACRO directive defines the end of a Macro definition. The directive does not take any parameters. See the MACRO directive for more information on defining Macros.

Syntax:

```
.ENDMACRO
```

Example:

```
.MACRO   SUBI16                 ; Start macro definition
         subi    r16,low(@0)    ; Subtract low byte
         sbci    r17,high(@0)   ; Subtract high byte
.ENDMACRO                       ; End macro definition
```

4.5.9 EQU - Set a symbol equal to an expression

The EQU directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the EQU directive is a constant and can not be changed or redefined.

Syntax:

```
.EQU label = expression
```

Example:

```
.EQU io_offset = 0x23
.EQU porta = io_offset + 2
.CSEG                           ; Start code segment
         clr     r2             ; Clear register 2
         out     porta,r2       ; Write to Port A
```

4.5.10 ESEG - EEPROM Segment

The ESEG directive defines the start of an EEPROM Segment. An Assembler file can consist of several EEPROM Segments, which are concatenated into one EEPROM Segment when assembled. The BYTE directive can not be used within an EEPROM Segment. The EEPROM Segments have their own location counter which is a byte counter. The ORG directive (see description later in this document) can be used to place constants at specific locations in the EEPROM memory. The directive does not take any parameters.

Syntax:

```
.ESEG
```

Example:

```
.DSEG                           ; Start data segment
       vartab:  .BYTE 4         ; Reserve 4 bytes in SRAM
.ESEG
       eevar:   .DW 0xff0f      ; Initialize one word in
                                ; EEPROM
.CSEG                           ; Start code segment
       const:   .DW 2           ; Write 0x0002 in prog.mem.
       mov r1,r0                ; Do something
```

4.5.11 EXIT - Exit this file

The EXIT directive tells the Assembler to stop assembling the file. Normally, the Assembler runs until end of file (EOF). If an EXIT directive appears in an included file, the Assembler continues from the line following the INCLUDE directive in the file containing the INCLUDE directive.

Syntax:

```
.EXIT
```

Example:

```
.EXIT                           ; Exit this file
```

4.5.12 INCLUDE - Include another file

The INCLUDE directive tells the Assembler to start reading from a specified file. The Assembler then assembles the specified file until end of file (EOF) or an EXIT directive is encountered. An included file may itself contain INCLUDE directives.

Syntax:

```
.INCLUDE “filename”
```

Example:

```
                                ; iodefs.asm:
.EQU    sreg=0x3f               ; Status register
.EQU    sphigh=0x3e             ; Stack pointer high
.EQU    splow=0x3d              ; Stack pointer low
                                ; incdemo.asm
.INCLUDE “iodefs.asm”           ; Include I/O definitions
       in    r0,sreg            ; Read status register
```

4.5.13 LIST - Turn the listfile generation on

The LIST directive tells the Assembler to turn listfile generation on. The Assembler generates a listfile which is a combination of assembly source code, addresses and opcodes. Listfile generation is turned on by default. The directive can also be used together with the NOLIST directive in order to only generate listfile of selected parts of an assembly source file.

Syntax:

```
.LIST
```

Example:

```
.NOLIST                         ; Disable listfile generation
.INCLUDE “macro.inc”            ; The included files will not
.INCLUDE “const.def”            ; be shown in the listfile
.LIST                           ; Reenable listfile generation
```

4.5.14 LISTMAC - Turn macro expansion on

The LISTMAC directive tells the Assembler that when a macro is called, the expansion of the macro is to be shown on the listfile generated by the Assembler. The default is that only the macro-call with parameters is shown in the listfile.

Syntax:

```
.LISTMAC
```

Example:

```
.MACRO   MACX                   ; Define an example macro
         add     r0,@0          ; Do something
         eor     r1,@1          ; Do something
.ENDMACRO                       ; End macro definition
.LISTMAC                        ; Enable macro expansion
         MACX    r2,r1          ; Call macro, show expansion
```

4.5.15 MACRO - Begin macro

The MACRO directive tells the Assembler that this is the start of a Macro. The MACRO directive takes the Macro name as parameter. When the name of the Macro is written later in the program, the Macro definition is expanded at the place it was used. A Macro can take up to 10 parameters. These parameters are referred to as @0-@9 within the Macro definition. When issuing a Macro call, the parameters are given as a comma separated list. The Macro definition is terminated by an ENDMACRO directive.

By default, only the call to the Macro is shown on the listfile generated by the Assembler. In order to include the macro expansion in the listfile, a LISTMAC directive must be used. A macro is marked with a + in the opcode field of the listfile.

Syntax:

```
.MACRO macroname
```

Example:

```
.MACRO     SUBI16               ; Start macro definition
           subi    @1,low(@0)   ; Subtract low byte
           sbci    @2,high(@0)  ; Subtract high byte
.ENDMACRO                       ; End macro definition
.CSEG                           ; Start code segment
           SUBI16  0x1234,r16,r17; Sub.0x1234 from r17:r16
```

4.5.16 NOLIST - Turn listfile generation off

The NOLIST directive tells the Assembler to turn listfile generation off. The Assembler normally generates a listfile which is a combination of assembly source code, addresses and opcodes. Listfile generation is turned on by default, but can be disabled by using this directive. The directive can also be used together with the LIST directive in order to only generate listfile of selected parts of an assembly source file.

Syntax:

```
.NOLIST                         ; Enable listfile generation
```

Example:

```
.NOLIST                         ; Disable listfile generation
.INCLUDE “macro.inc”            ; The included files will not
.INCLUDE “const.def”            ; be shown in the listfile
.LIST                           ; Reenable listfile generation
```

4.5.17 ORG - Set program origin

The ORG directive sets the location counter to an absolute value. The value to set is given as a parameter. If an ORG directive is given within a Data Segment, then it is the SRAM location counter which is set, if the directive is given within a Code Segment, then it is the Program memory counter which is set and if the directive is given within an EEPROM Segment, then it is the EEPROM location counter which is set. If the directive is preceded by a label (on the same source code line), the label will be given the value of the parameter. The default values of the Code and EEPROM location counters are zero, whereas the default value of the SRAM location counter is 32 (due to the registers occupying addresses 0-31) when the assembling is started. Note that the EEPROM and SRAM location counters count bytes whereas the Program memory location counter counts words.

Syntax:

```
.ORG expression
```

Example:

```
.DSEG                           ; Start data segment
.ORG 0x67                       ; Set SRAM address to hex 67
       variable:.BYTE 1         ; Reserve a byte at SRAM
                                ; adr.67H
.ESEG                           ; Start EEPROM Segment
.ORG 0x20                       ; Set EEPROM location
                                ; counter
       eevar:   .DW 0xfeff      ; Initialize one word
.CSEG
.ORG 0x10                       ; Set Program Counter to hex
                                ; 10
       mov      r0,r1           ; Do something
```

4.5.18 SET - Set a symbol equal to an expression

The SET directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the SET directive can be changed later in the program.

Syntax:

```
.SET label = expression
```

Example:

```
.SET io_offset = 0x23
.SET porta = io_offset + 2
.CSEG                          ; Start code segment
       clr      r2             ; Clear register 2
       out      porta,r2       ; Write to Port A
```

4.6 Expressions

The Assembler incorporates expressions. Expressions can consist of operands, operators and functions. All expressions are internally 32 bits.

4.6.1 Operands

The following operands can be used:

- User defined labels which are given the value of the location counter at the place they appear.
- User defined variables defined by the SET directive
- User defined constants defined by the EQU directive
- Integer constants: constants can be given in several formats, including
  a) Decimal (default): 10, 255
  b) Hexadecimal (two notations): 0x0a, $0a, 0xff, $ff
  c) Binary: 0b00001010, 0b11111111
- PC - the current value of the Program memory location counter

4.6.2 Functions

The following functions are defined:

- LOW(expression) returns the low byte of an expression
- HIGH(expression) returns the second byte of an expression
- BYTE2(expression) is the same function as HIGH
- BYTE3(expression) returns the third byte of an expression
- BYTE4(expression) returns the fourth byte of an expression
- LWRD(expression) returns bits 0-15 of an expression
- HWRD(expression) returns bits 16-31 of an expression
- PAGE(expression) returns bits 16-21 of an expression
- EXP2(expression) returns 2^expression
- LOG2(expression) returns the integer part of log2(expression)

4.6.3 Operators

The Assembler supports a number of operators which are described here. The higher the precedence, the higher the priority. Expressions may be enclosed in parentheses, and such expressions are always evaluated before combined with anything outside the parentheses.

4.6.3.1 Logical Not

Symbol: !

Description: Unary operator which returns 1 if the expression was zero, and returns 0 if the expression was nonzero

Precedence: 14

Example:
```
ldi   r16,!0xf0      ; Load r16 with 0x00
```

4.6.3.2 Bitwise Not

Symbol: ~

Description: Unary operator which returns the input expression with all bits inverted

Precedence: 14

Example:
```
ldi   r16,~0xf0      ; Load r16 with 0x0f
```

4.6.3.3 Unary Minus

Symbol: -

Description: Unary operator which returns the arithmetic negation of an expression

Precedence: 14

Example:
```
ldi   r16,-2         ; Load -2(0xfe) in r16
```

4.6.3.4 Multiplication

Symbol: *

Description: Binary operator which returns the product of two expressions

Precedence: 13

Example:
```
ldi   r30,label*2    ; Load r30 with label*2
```

4.6.3.5 Division

Symbol: /

Description: Binary operator which returns the integer quotient of the left expression divided by the right expression

Precedence: 13

Example:
```
ldi   r30,label/2    ; Load r30 with label/2
```

4.6.3.6 Addition

Symbol: +

Description: Binary operator which returns the sum of two expressions

Precedence: 12

Example:
```
ldi   r30,c1+c2      ; Load r30 with c1+c2
```

4.6.3.7 Subtraction

Symbol: -

Description: Binary operator which returns the left expression minus the right expression

Precedence: 12

Example:
```
ldi   r17,c1-c2      ;Load r17 with c1-c2
```

4.6.3.8 Shift left

Symbol: <<

Description: Binary operator which returns the left expression shifted left a number of times given by the right expression

Precedence: 11

Example:
```
ldi   r17,1<<bitmask ;Load r17 with 1 shifted
                     ;left bitmask times
```

4.6.3.9 Shift right

Symbol: >>

Description: Binary operator which returns the left expression shifted right a number of times given by the right expression.

Precedence: 11

Example:
```
ldi   r17,c1>>c2     ;Load r17 with c1 shifted
                     ;right c2 times
```

4.6.3.10 Less than

Symbol: <

Description: Binary operator which returns 1 if the signed expression to the left is Less than the signed expression to the right, 0 otherwise

Precedence: 10

Example:
```
ori   r18,bitmask*(c1<c2)+1      ;Or r18 with
                                 ;an expression
```

4.6.3.11 Less or Equal

Symbol: <=

Description: Binary operator which returns 1 if the signed expression to the left is Less than or Equal to the signed expression to the right, 0 otherwise

Precedence: 10

Example:
```
ori   r18,bitmask*(c1<=c2)+1     ;Or r18 with
                                 ;an expression
```

4.6.3.12 Greater than

Symbol: >

Description: Binary operator which returns 1 if the signed expression to the left is Greater than the signed expression to the right, 0 otherwise

Precedence: 10

Example:
```
ori   r18,bitmask*(c1>c2)+1      ;Or r18 with
                                 ;an expression
```

4.6.3.13 Greater or Equal

Symbol: >=

Description: Binary operator which returns 1 if the signed expression to the left is Greater than or Equal to the signed expression to the right, 0 otherwise

Precedence: 10

Example:
```
ori   r18,bitmask*(c1>=c2)+1     ;Or r18 with
                                 ;an expression
```

4.6.3.14 Equal

Symbol: ==

Description: Binary operator which returns 1 if the signed expression to the left is Equal to the signed expression to the right, 0 otherwise

Precedence: 9

Example:
```
andi  r19,bitmask*(c1==c2)+1     ;And r19 with
                                 ;an expression
```

4.6.3.15 Not Equal

Symbol: !=

Description: Binary operator which returns 1 if the signed expression to the left is Not Equal to the signed expression to the right, 0 otherwise

Precedence: 9

Example:
```
.SET flag=(c1!=c2)       ;Set flag to 1 or 0
```

4.6.3.16 Bitwise And

Symbol: &

Description: Binary operator which returns the bitwise And between two expressions

Precedence: 8

Example:
```
ldi   r18,High(c1&c2)   ;Load r18 with an expression
```

4.6.3.17 Bitwise Xor

Symbol: ^

Description: Binary operator which returns the bitwise Exclusive Or between two expressions

Precedence: 7

Example:
```
ldi   r18,Low(c1^c2)    ;Load r18 with an expression
```

4.6.3.18 Bitwise Or

Symbol: |

Description: Binary operator which returns the bitwise Or between two expressions

Precedence: 6

Example:
```
ldi   r18,Low(c1|c2)    ;Load r18 with an expression
```

4.6.3.19 Logical And

Symbol: &&

Description: Binary operator which returns 1 if the expressions are both nonzero, 0 otherwise

Precedence: 5

Example:
```
ldi   r18,Low(c1&&c2)   ;Load r18 with an expression
```

4.6.3.20 Logical Or

Symbol: ||

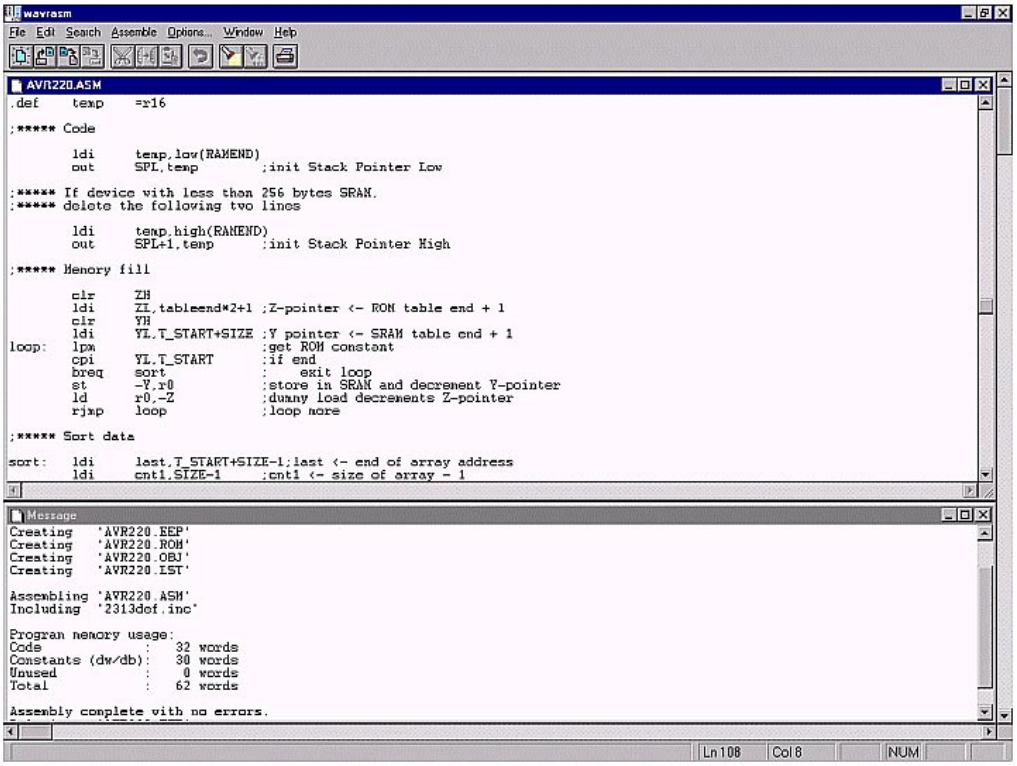
Description: Binary operator which returns 1 if one or both of the expressions are nonzero, 0 otherwise

Precedence: 4

Example:
```
ldi   r18,Low(c1||c2)   ;Load r18 with an expression
```

4.7 Microsoft Windows specifics

This section describes the features specific to WAVRASM. Only the menu items specific to the Assembler are described. It is assumed that the user is familiar with the “Search” and “Window” menu items. A typical editing session with the Assembler is shown in the following figure.



4.7.1 Opening Assembly Files

A new or existing assembly files can be opened in WAVRASM. Theoretically there is no limit on how many assembly files which can be open at one time. The size of each file must be less than about 28K bytes due to a limitation in MS-Windows. It is still possible to assemble files larger than this, but they can not be edited in the integrated editor. A new editor window is created for every assembly file which is opened.

To create a new assembly file click the button on the toolbar or choose “File → New” (ALT-F N) from the menu. To open an existing file click the button on the toolbar or choose “File → Open” (ALT-F O) from the menu.

4.7.2 The Integrated Editor

When WAVRASM is finished loading a file, the text editor will be inactive. Refer to the section on opening files on how to open a file. Right after a file is loaded into an editor window of the Assembler, the insertion point appears in the upper left corner of the window.

4.7.3 Typing and Formatting Text

The insertion point moves to the right when typing. If text is written beyond the right margin, the text automatically scrolls to the left so that the insertion point is always visible.

4.7.4 Moving the Insertion Point

The insertion point can be moved anywhere by moving the mouse cursor to the point where the insertion point is wanted and click the left button.

To move the insertion point with the keyboard, use the following keys or key combinations:

To move the insertion point:	Press:
to the right in a line of text	Right arrow key
to the left in a line of text	Left arrow key
up in a body of text	Up arrow key
down in a body of text	Down arrow key
to the beginning of a line of text	Home
to the end of a line of text	End
to the beginning of the file	Ctrl+Home
to the end of the file	Ctrl+End

4.7.5 Formatting Text

The keys in the table below describes the necessary operations to type in the text exactly as wanted.

To:	Press:
insert a space	Spacebar
delete a character to the left	Backspace
delete a character to the right	Del
end a line	Enter
indent a line	Tab
insert a tab stop	Tab

To split a line, move the insertion point to the position where the break is wanted and press Enter.

To join two lines, move the insertion point to the beginning of the line to move, and press Backspace. The editor joins the line with the line above.

4.7.6 Scrolling

If a line of text is longer or wider than can be shown at one time, the file can be scrolled by using the scroll bars.

4.7.7 Editing Text

The Edit-menu contains some functions which can be of much help in editing. Text can be deleted, moved or copied to new locations. The Undo command can be used to revert the last edit. Transferring text to and from other windows or applications can be done via the clipboard. When text is deleted or copied with the commands Cut or Copy, the text is placed in the Clipboard. The Paste command copies text from the Clipboard to the editor.

4.7.8 Selecting Text

Before a command is selected from the Edit-menu to edit text, the text to operate on must first be selected.

Selecting text with the keyboard:

1. Use the arrow keys to move the insertion point to the beginning of the text to select.
2. Press and hold the Shift-key while moving the insertion point to the end of the text to select. Release the Shift-key. To cancel the selection, press one of the arrow keys.

Selecting text with the mouse:

1. Move the mouse cursor to the beginning of the text to select.
2. Hold down the left mouse button while moving the cursor to the end of the text to select. Release the mouse button.
3. To cancel the selection, press the left mouse button or one of the arrow keys.

4.7.9 Replacing Text

When text is selected, it can be immediately replaced it by typing new text. The selected text is deleted when the first new character is typed.

Replacing text:

1. Select the text to replace.
2. Type the new text.

Deleting Text:

1. Select the text to delete.
2. Press the Del key.

To restore the deleted text, press the key on the toolbar or choose “Edit → Undo” (Alt+Backspace) from the menu immediately after deleting the text.

4.7.10 Moving Text

Text can be moved from one location in the editor by first copy the text to the Clipboard with the Cut command, and then pasting it to its new location using the Paste command.

To move text:

1. Select the text to move.
2. Press the button on the toolbar or choose “Edit → Cut” (Shift+Del) from the menu. The text is placed in the Clipboard.
3. Move the insertion point to the new location.
4. Press the button on the toolbar or choose “Edit → Paste” (Shift+Ins) from the menu.

4.7.11 Copying Text

If some text will be used more than once, it need not be typed each time. The text can be copied to the Clipboard with Copy, and can then be pasted in many places by using the Paste command.

To copy text:

1. Select the text to copy.
2. Click the button on the toolbar or choose “Edit → Copy” (Ctrl+Ins) from the menu. The text is placed in the Clipboard.
3. Move the insertion point to the location to place the text.
4. Click the button on the toolbar or choose “Edit → Paste” (Shift-Ins) from the menu.

4.7.12 Undoing an Edit

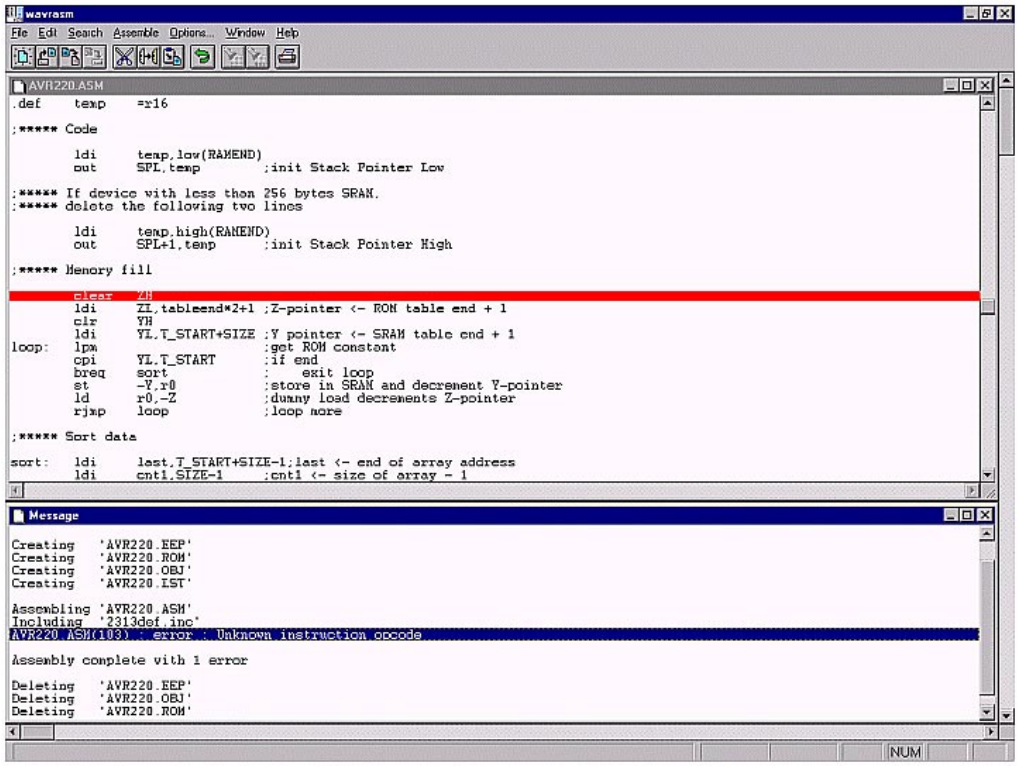
The Undo command can be used to cancel the last edit. For example, text may accidentally have been deleted, or it has been copied to a wrong location. If the Undo command is chosen immediately after the mistake was done, the text will be restored to what it was before the mistake.

To undo the last edit click the button on the toolbar or choose “Edit → Undo” (Alt+Backspace) from the menu.

4.7.13 Click On Errors

The Assembler has a click on error function. When a program is assembled, a message window appears on the screen. If errors are encountered, the errors are listed in this message window. If one of the error lines in the message window is clicked, the source line turns inverted red. If the error is in a included file, nothing happens.

This feature is demonstrated in the following figure:

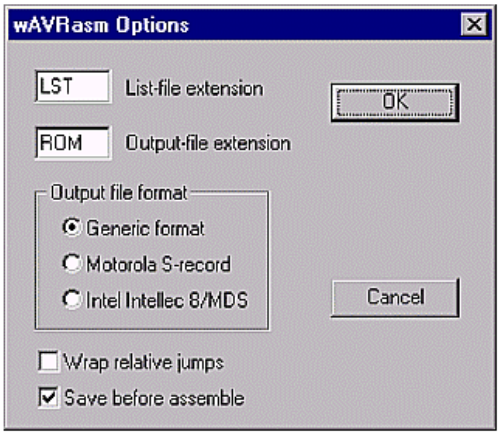


If the message window line is doubleclicked, the file containing the error becomes the active window, and the cursor is placed at the beginning of the line containing the error. If the file containing the error is not opened (for instance an included file), then the file is automatically opened.

Note that this function points to lines in the assembled file. This means that if lines are added or removed in the source file, the file must be reassembled in order to get the line numbers right.

4.7.14 Setting Program Options

Some of the default values of WAVRASM can be changed in the options menu. If “Options” is selected on the menu bar, the following dialog box pops up.



In the box labeled “List-file extension” the default extension on the list file(s) is written, and in the box labeled “Output-file extension” the default extension of the output file is written. In the box labeled “Output file format” the type of format wanted on the output file can be selected. If the OK button is clicked, the values are remembered in subsequent runs of the Assembler. Note that the object file (used by the simulator) is not affected by these options; the extension of the object file is always ‘OBJ’ and the format is always the same. If an EEPROM Segment has been defined in the code, the assembler will also generate a file with extension ‘EEP’ which is the initial values for the EEPROM memory. This EEPROM initialization file is in the same format as the Output file format selected.

The “Wrap relative jumps” option tells the Assembler to use wrapping of addresses. This feature is only useful when assembling for devices with 4K words of program memory. Using this option on such devices, the relative jump and call instructions will reach the entire program memory.

The “Save before assemble” option makes the Assembler automatically save the contents of the editor before assembling is done.

4.8 Command line version

For the MS-DOS command line version the Assembler is invoked by the command

```
AVRASM [-m | -i | -g][-w] input.asm output.lst output.rom
```

AVRASM will now read source from input.asm, produce the listfile output.lst, output.rom and the object file input.obj. The objectfile '*.obj' is used by the MS-Windows simulator.

The user can select which output format to generate by using one of the options -m (Motorola S-record), -i (Intel Hex) or -g (Generic). The Generic file format is used by default.

The -w option tells the Assembler to use wrapping of addresses. This feature is only used when assembling for devices with 4K words of program memory. Using this switch on these devices, the relative jump and call instructions will reach the entire program memory.