

The graphical environment for the software development of microcontrollers with AVR architecture

"Algorithm Builder"

This tool covers the complete development cycle, from the input algorithm to the in-circuit programming of the chip. It features the assembler level and macro level programming, debugging and operations with signed values including diversity formats. This development platform can be compared with a high-level programming language.

Its graphic interface represents a powerful tool for programmers. The program can be entered as a flow-chart, which allows both the experienced programmers and the novices to create complex algorithms in a quick and facile way. This environment enables to reduce the development time up to 3-5 X.

This tool runs under Windows 95, 98, NT, ME and XP OS platforms and requires of the "Courier" font.

Basis of algorithm

Any software can be divided into several logically accomplished fragments. Usually, the final operator of these fragments is an unconditional jump or a subprogram return, i.e. the operators, after which the linear program executing is unambiguously terminated.


The arbitrarily allocation of these logically accomplished blocks and setup of conditional and unconditional interconnections builds up the algorithm.

The elements of algorithm

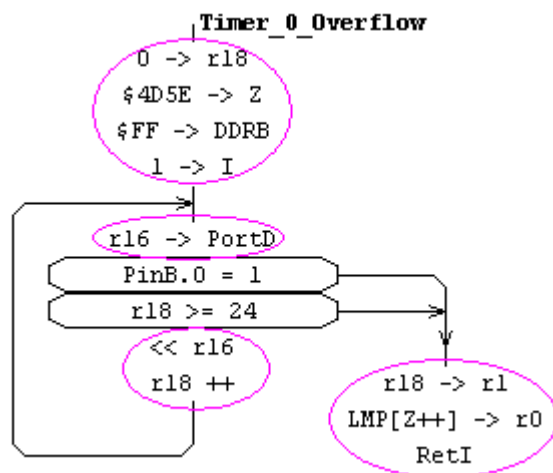
The Algorithm Builder contains seven basic elements:

"FIELD", "LABEL", "VERTEX", "CONDITION", "JMP Vector", "SETTER", and "TEXT".


The "FIELD" element

An aligned in a block string represents this element. It is destined for realization of most commands of microcontroller. To insert a new FIELD, select the "Elements\Field" menu item, click on the  button, press the "Alt+F" keys, or press the "Enter" key (if the editor focus is not in the "TEXT" elements).

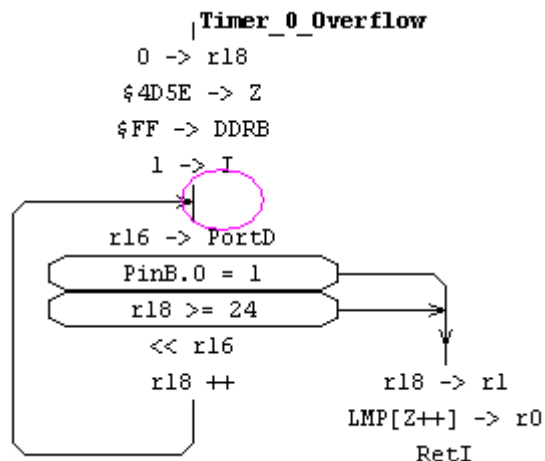
The following examples of "FIELD" -elements are encircled:



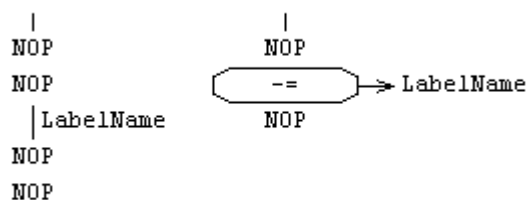
The "LABEL" element

A short vertical line represents the label, which is located in the operators' block. An optional name can be placed at the left or right side of this line. Labels can be used as a reference for conditional and unconditional branches. To insert a new LABEL, select the "Elements\Label" menu item, click on the  button, or press the "Alt+L" keys.

In order to assign a physical address to a label, type the constant or algebraic expression before the Label name (if it exist). To swap the position of the name to either side of the short line, press the “Tab” key.
The following example of a “LABEL”-element is encircled:




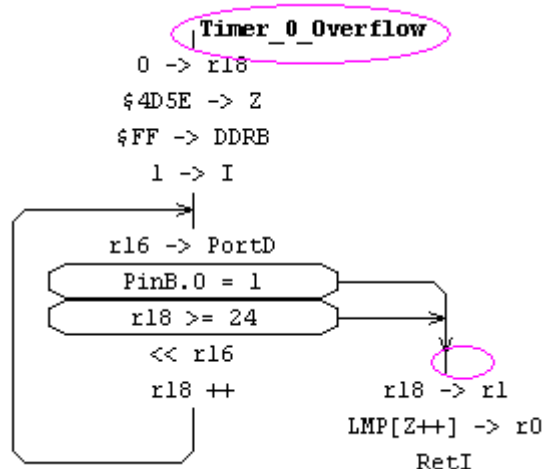
The branch vectors can point to a label in two ways. In the first way, they can be connected *directly* to a label as shown above. In this case, the declaration of a label name is not necessary. In the second way, they can be connected by a label *name* as shown below:



The label name is a *constant* with a physical address of a point in the program memory.

The “VERTEX” element (vertex of a block)

The “VERTEX”-element is very similar to the “LABEL”. But unlike the “LABEL”, the “VERTEX” defines the position of entire block of operators. To insert a new VERTEX, select the “Elements\Vertex” menu item, click on the  button, press the “Alt+V” keys, or click the left mouse button with “Alt+Ctrl+Shift” keys, on free space.
The following examples of “VERTEX”-element are encircled.



As a rule, a vertex name is only declared at the beginning of subprograms or macros.

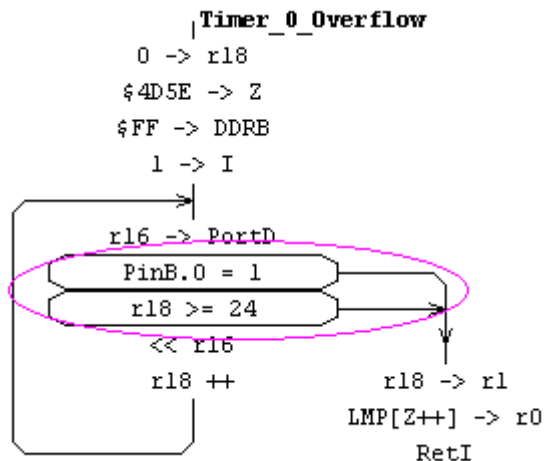
The “CONDITION” element (conditional branch)

This element executes a conditional branch. It is displayed by an oval contour containing a branch condition and a branch vector with an arrow at the end, where the optional label name may be typed in. The unnamed vector arrow must point to a “LABEL”, a “VERTEX”, or a segment of another vector.

To move the vector to a desired point, click the left mouse button along with the pressed “Alt” key. To edit the vector, use the arrow keys along with the “Alt” key. For toggling the editing of condition and the vector name, press the “Tab” key.

To insert a new “CONDITION”, select the “Elements\Condition” menu item, click on the  button, or press “Alt+C” keys.

The following examples of the “CONDITION” elements are encircled:

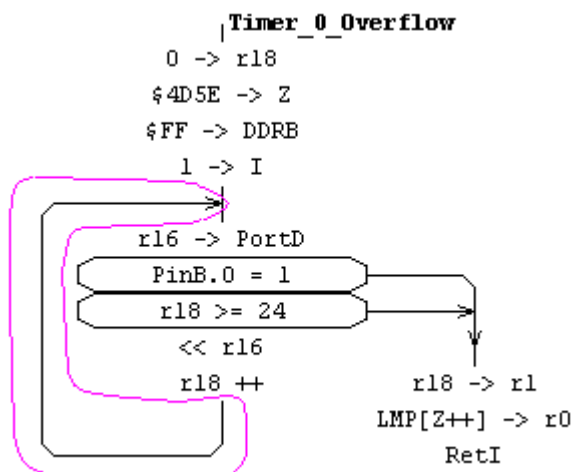


The “JMP Vector”-element (unconditional branch)

This element executes the short relative unconditional branch (analogue: “RJMP” in a classic assembler). It is represented by a broken line with an arrow at the end starting from the middle of a block as shown in the following example.

To insert a new “JMP Vector”, select the “Elements\JMP Vector” menu item, click the  button, or press the “Alt+J” keys.

The following example of a “JMP Vector” element is encircled:



The “SETTER” element

This element is displayed by a grayed rectangle with a name of an adjustable component of a microcontroller, such as ADC, Timer\Counter etc. The “SETTER” serves for configure the microcontroller’s operations for the initialization or modification of the appropriate control registers.

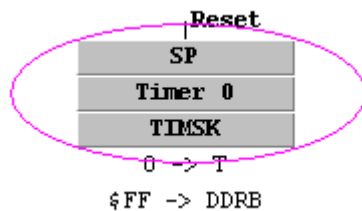
Before using the “SETTER”, the type of the microcontroller must be determined (the “Options\Project options” menu item, at the “Chip” table).

To insert a new “SETTER”, select the “Elements\Setter” menu item, click the **S** button, or press the “Alt+S” keys. To edit an existing element, double-click on it, or press the “Shift+Enter” keys.

The “SETTER”- element is a macro-operator. After the compilation, the element will be translated to a sequence of microcontroller operations, which will load necessary constants to the appropriate control registers. Note the working register r16 is used for these operations.

When adjusting some components, such as ADC or UART, the “SETTER” may modify a few I/O registers. In this case, it is possible to appoint the effect on specific registers.

The following examples of “SETTER”-elements are encircled:



An example of the ADC setter window for the ATmega128:

The set of operations, generated by this SETTER is found rightmost (underneath the buttons). Alongside, in vertical direction, there are check boxes for the selective activation of the appropriate I/O registers.

At right side of the sheet there is a vertical series of check boxes for activation the used I/O registers. It allows activating of registers selectively.

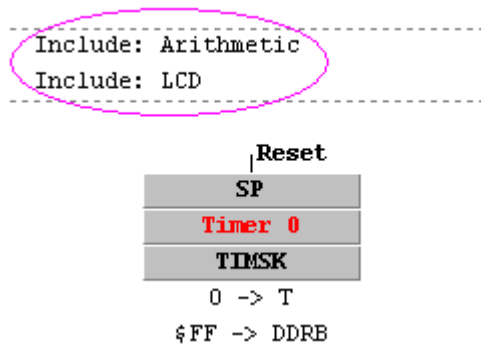
Some of check boxes have an "exclusive" option (in the current example, the "SFIO" register is not used by ADC alone). If the “exclusive” option is marked, the SETTER will clear the unutilized bits of the component. In case of “exclusive” option is left unmarked, these bits will be unchanged but in this case the program code will be a little longer. This guarantees, that the setting of another component, which uses this register too, will not be crushed.

The “TEXT”- element (line of local text editor)

The text line begins from the left of operating space of an algorithm and represents this element. The local text editor is composed by a number of such lines. These lines are used for writing down the some compiler directives or comments.

To insert a new text line, select the “Elements\Text” menu item, click the **T** button, or press the “Alt+T” keys. Comments have to start with two backslashes: “//”.

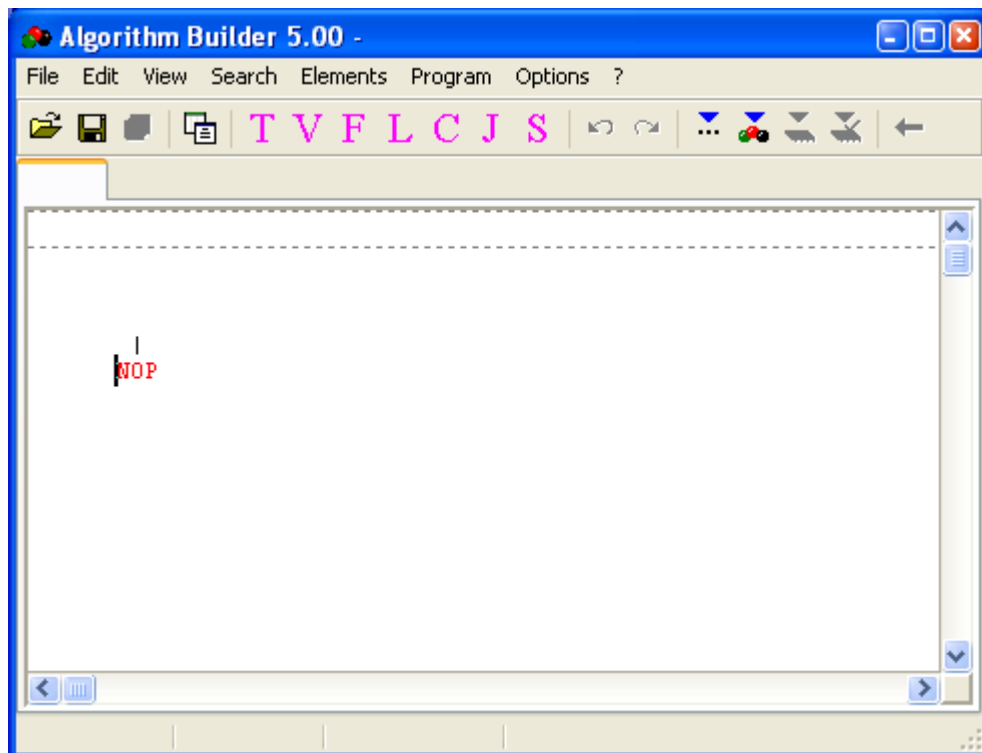
An encircled example of the “TEXT”-elements is below:



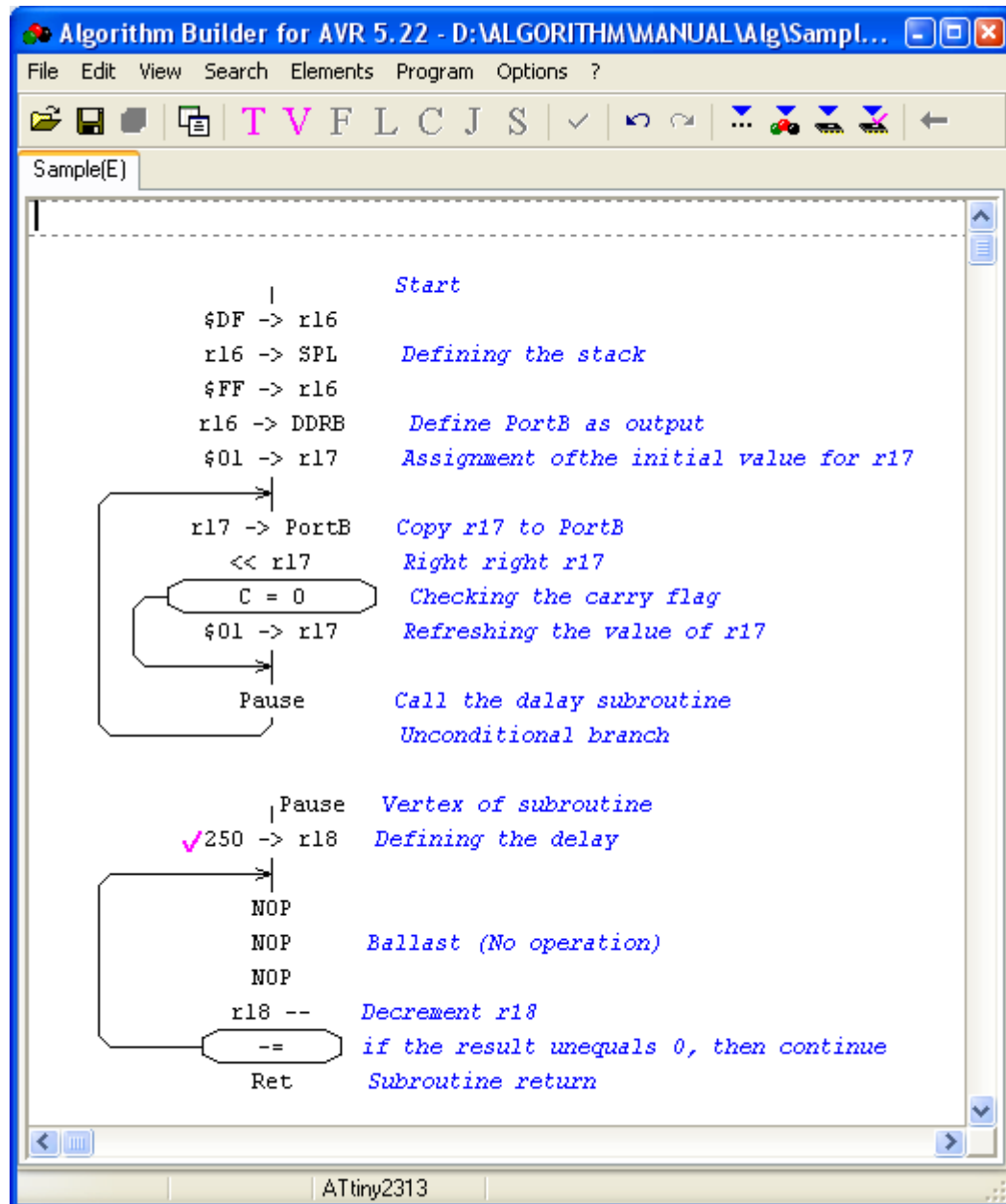
The example of an elementary algorithm

In case another algorithm project is open, it has to be exited. To do this, select the “File\Close project” menu item.

To create a new project, select the “File\New” menu item whereupon, the first tab with elements “TEXT”, “VERTEX” and “FIELD” with a void operator “NOP” will appear on the operating field:



Type in the following example of elementary algorithm (without comments):



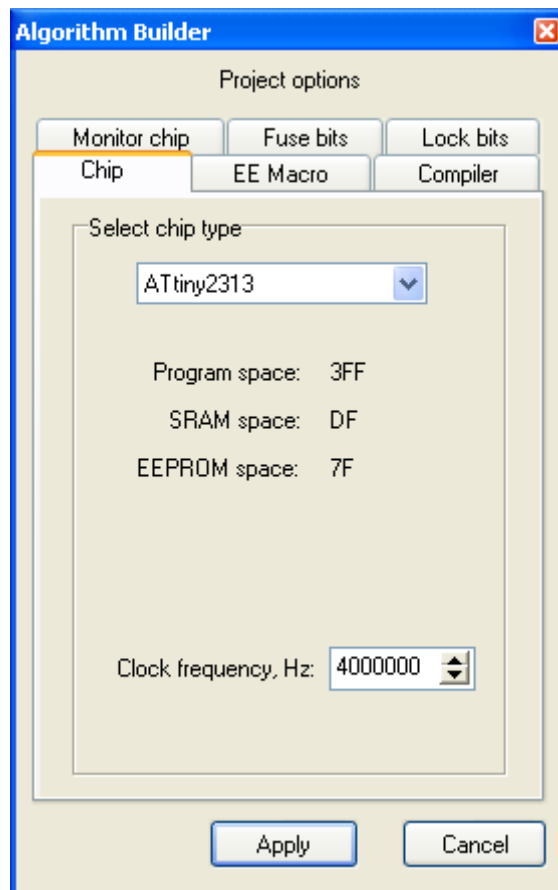
This algorithm alternately sets the bits of the port B. The algorithm contains only the elementary instructions (operations) of the microcontroller. This example can be found in the "Examples\Simple" directory, but it is preferably to type in manually.


For the "copy" operations, arrow symbols ">" are frequently used. For your convenience, they can be typed in by one key with the "~" symbol (under the "Escape" key). To insert a next "FIELD"- element, press the "Enter"-key, to insert a "LABEL" – "Alt+L", "CONDITION" – "Alt+C" etc. Alternatively, you can click on the appropriate buttons in the tool bar.

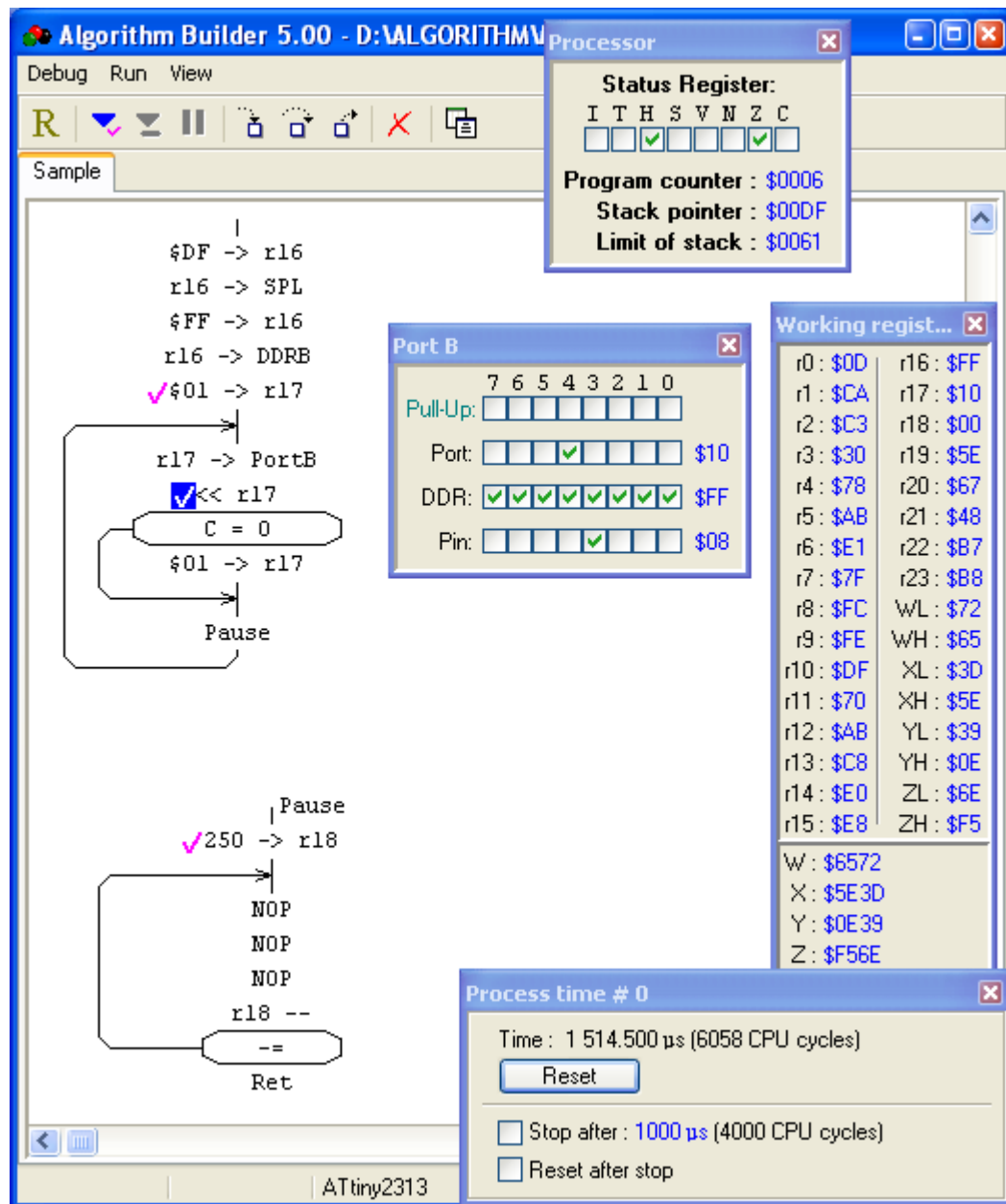
To insert a new "VERTEX"- element, put the editor focus on the final element of previous block and press the "Alt+V" keys.

The ✓ mark at the operator "250->r18" means the debugger breakpoint. To insert or delete such mark, put the cursor upon the appropriate operator and press the "F5" key.

Before the compilation, it is necessary to define the microcontroller type with the "Options\Project options..." menu item:



To run the algorithm in the simulation mode, press “F9” key or click the  button. At first, the algorithm will be compiled next, in case of absence of errors, simulation will start. To open the appropriate windows, use the “View” menu. To observe the process in the algorithm, it is enough to open the windows: “PortB”, “Processor”, and “Working registers”:



The ☒ mark indicated the current state of the program counter. This mark is close by an operator which will be executed in next step.

After a start of the simulator, accidental values are stored in the working registers and SRAM, like in a real microcontroller after the powering.

For the step-by-step executing of algorithm with entrance to subprograms, use the “F7” key or button.

For the step-by-step executing without entrance to subprogram, use the “F8” key or button.

For executing up to return from subprogram, use the “F6” key or button.

To start a continuous execution up to breakpoint (), press the “F9” key or button.

To start a continuous execution up to selected element, press the “F4” key or button.

In both cases, the execution can be interrupted by pressing of the “F2” key or button.

During the execution of this algorithm, you can watch the sequential setting of logical one to the bits of port B. You may program this algorithm into the real chip and observe the execution by means of an oscilloscope.

The operators of the microcontrollers

The AVR instruction set comprises of more than 130 elementary operations. The following table contains all these instructions.

The copying and arithmetic-logic operators

The “FIELD” elements are used for these operators.

Template	Operation	Example	Equivalent
R -> R	Copying working register to working register	R0 -> R1	MOV R,R
# -> R	Loading immediate # constant to working register	24 -> r16	LDI R,K
[X] -> R	Indirect copying from SRAM to working register	[X] -> R2	LD R,X
[X++] -> R	Indirect copying from SRAM to working register and post-increment	[X++] -> R3	LD R,X+
[--X] -> R	Pre-decrement and indirect copying from SRAM to working register	[--X] -> R4	LD R,-X
[Y] -> R	Indirect copying from SRAM to working register	[Y] -> R5	LD R,Y
[Y++] -> R	Indirect copying from SRAM to working register and post-increment	[Y++] -> R6	LD R,Y+
[--Y] -> R	Pre-decrement and indirect copying from SRAM to working register	[--Y] -> R7	LD R,-Y
[Y+#] -> R	Indirect copying with # displacement from SRAM to working register	[Y+2] -> r8	LDD R,Y+q
[Z] -> R	Indirect copying from SRAM to working register	[Z] -> R9	LD R,Z
[Z++] -> R	Indirect copying from SRAM to working register and post-increment	[Z++] -> R10	LD R,Z+
[--Z] -> R	Pre-decrement and indirect copying from SRAM to working register	[--Z] -> R11	LD R,-Z
[Z+#] -> R	Indirect copying with # displacement from SRAM to working register	[Z+1] -> R12	LDD R,Z+q
[#] -> R	Direct copying by # address from SRAM to working register	[\$60] -> r14	LDS R,k
R -> [X]	Copying working register indirect to SRAM	R15 -> [X]	ST X,R
R -> [X++]	Copying working register indirect to SRAM and post-increment	R16 -> [X++]	ST X+,R
R -> [--X]	Pre-decrement and copying working register indirect to SRAM	R17 -> [--X]	ST -X,R
R -> [Y]	Copying working register indirect to SRAM	R18 -> [Y]	ST Y,R
R -> [Y++]	Copying working register indirect to SRAM and post-increment	R19 -> [Y++]	ST Y+,R
R -> [--Y]	Pre-decrement and copying working register indirect to SRAM	R20 -> [--Y]	ST -Y,R
R -> [Y+#]	Copying working register indirect with # displacement to SRAM	R21 -> [Y+2]	STD Y+q,R
R -> [Z]	Copying working register indirect to SRAM	R22 -> [Z]	ST Z,R
R -> [Z++]	Copying working register indirect to SRAM and post-increment	R23 -> [Z++]	SR Z+,R
R -> [--Z]	Pre-decrement and copying working register indirect to SRAM	R24 -> [--Z]	ST -Z,R
R -> [Z+#]	Copying working register indirect with # displacement to SRAM	R25 -> [Z+3]	STD Z+q,R
R -> [#]	Copying working register directly by # address to SRAM	R26 -> [\$200]	STS k,R
LPM LPM[Z]	Copying from program memory by address Z to r0		LPM

LPM -> R LPM[Z] -> R	Copying from program memory by address Z to working register		LPM R,Z
LPM[Z++] -> R	Copying from program memory by address Z to working register and post-increment	LPM[Z++] -> r5	LPM R,Z+
SPM	Store program memory		SPM
P -> R	Copying I/O register to working register	P\$19 -> WH	IN R,P
R -> P	Copying working register to I/O register	XL -> PortB	OUT P,R
R->	Copying working register to stack	r0->	PUSH R
->R	Copying from stack to working register	->r1	POP R
NOP	No operation		NOP
R + R	Add right working register to left working register	r0 + r1	ADD R,R
R + R +	Add with carry right working register to left working register	R0 + R5 +	ADC R,R
RR + #	Add # constant to double working register	W + 32	ADIW RWl,K
R - R	Subtract right working register from the left working register	r4 - r5	SUB R,Rr
R - #	Subtract # constant from the working register	XL - 2	SUBI R,K
R + #	Add # constant to the working register	R17 + 12	SUBI R,256-K
R - R -	Subtract right working register from the left working register with carry	r5 - r6 -	SBC R,R
R - # -	Subtract # constant from the working register with carry	ZL - \$70 -	SBCI R,K
RR - #	Subtract # constant from the double working register (W, X, Y, Z)	X - \$2E	SBIW RWl,K
R & R	Bit operation AND between working registers	r7 & r8	AND R,R
R & #	Bit operation AND between working register and # constant	r17 & #b00111101	ANDI R,K
R & #	Bit operation AND between working register and inverted # constant	r18 & #b00010011	CBR R,K
R ! R	Bit operation OR between working registers	R9 ! r10	OR R,R
R ! #	Bit operation OR between working register and # constant	R18 ! #o147	ORI R,K
R ^ R	Bit operation EXCLUSIVE OR between working registers	R11 ^ r12	EOR R,R
R	Bit inversion of working register	-r8-	COM R
-R	Arithmetic inversion of working register	-RB	NEG R
R++	Increment working register	Count++	INC R
R--	Decrement working register	Idx--	DEC R
R?	Test working register (R & R)	R16?	TST R
^R	Clear working register (R ^ R)	R8	CLR R
R * R	Unsigned multiply two working registers.	R6 * R17	MUL R,R
±R * ±R	Signed multiply two working registers	±r16 * ±r20	MULS R,R
±R * R R * ±R	Multiply signed working registers and unsigned working register	±r18 * r22 r22 * ±r18	MULSU R,R
<<(R * R)	Fractional unsigned multiply two working registers	<<(R6 * R17)	FMUL R,R
<<(±R * ±R)	Fractional signed multiply two	<<(±r16 * ±r20)	FMULS R,R

	working registers		
<<(±R * R) <<(R * ±R)	Fractional multiply unsigned working registers and signed working register	<<(±r18 * r22) <<(r22 * ±r18)	FMULSU R,R
1 -> P.#	Set # bit in I/O register	1 -> DDB2	SBI P,b
0 -> P.#	Clear # bit in I/O register	0 -> PortD.7	CBI P,b
<<R	Logical shift left of working register	<<r18	LSL R
R>>	Logical shift right of working register	R19>>	LSR R
<<R<	Logical shift left of working register trough carry	<<r20<	ROL R
>R>>	Logical shift right of working register trough carry	>r21>>	ROR R
±R>>	Arithmetic shift right of working register	±r22>>	ASR R
>>R<<	Swap nibbles in working register	>>r23<<	SWAP R
R.#->	Copy bit # of working register to bit T of SREG	RA.6->	BST R,b
->R.#	Copy bit T of SREG to the # bit of working register	->r0.4	BLD R,b
# -> R.#	Write # (0 or 1) to # bit of working register	0 -> r18.4 1 -> r20.0	ANDI R,256-K ORI R,K
RR -> RR	Copy double working register to double working register	Y -> X	MOVW R,R
1 -> C	Set carry		SEC
0 -> C	Clear carry		CLC
1 -> N	Set negative flag		SEN
0 -> N	Clear negative flag		CLN
1 -> Z 1 -> .Z	Set zero flag		SEZ
0 -> Z 0 -> .Z	Clear zero flag		CLZ
1 -> I	Global interrupt enable		SEI
0 -> I	Global interrupt disable		CLI
1 -> S	Set signed test flag		SES
0 -> S	Clear signed test flag		CLS
1 -> V	Set twos complement overflow		SEV
0 -> V	Clear twos complement overflow		CLV
1 -> T ; 1->	Set bit T in SREG		SET
0 -> T ; 0->	Clear bit T in SREG		CLT
1 -> H	Set half carry flag in SREG		SEH
0 -> H	Clear half carry flag in SREG		CLH
SLEEP	Sleep		SLEEP
WDR	Watchdog reset		WDR
JMP #	Direct jump	JMP LabelName	JMP k
#/	Direct subprogram call	LabelName/	CALL k
JMP[Z]	Indirect jump by Z		IJMP
#	Relative subprogram call	LabelName	RCALL k
CALL[Z]	Indirect subprogram call by Z	CALL	ICALL
RET	Subprogram return	RET	RET
RETI	Subprogram call and enable global interrupt	RETI	RETI
R = R	Compare two working registers	R2 = R5	CP R,R
R = R =	Compare with carry two working registers	R6 = R5 =	CPC R,R
R = #	Compare working register and # constant	YL = 47	CPI R,K

The results of the multiplication operations are placed in the double register **R0,R1**.

Note, the operations between a working register and constant (for example “#->R” or “R & #”) work with **r16** to **r31** only. The bit operations (for example “1->PortB.4”) work with \$00 to \$1F address of I/O registers only.

The operators of conditional branches

For these operators the “CONDITION”-elements are used. The following conditions can be entered into an oval:

Template	Comments	Example	Equivalent
=	Result is equal to 0 (Z=1)		BREQ k
!=	Result is not equal to 0 (Z=0)		BRNE k
C = 1			BRCS k
C = 0			BRCC k
>=	Greater or equal		BRSH k
<	Less		BRLO k
-	Negative result		BRMI k
+	Positive result		BRPL k
±>=	Signed greater or equal		BRGE k
±<	Signed less		BRLT k
H = 1			BRHS k
H = 0			BTHC k
T = 1			BRTS k
T = 0			BRTC k
V = 1			BRVS k
V = 0			BRVC k
I = 1			BRIE k
I = 0			BRID k

The vector of the applied “CONDITION”-element has to end with a “LABEL” or “VERTEX”, at a segment of another vector, or to point to a label name.

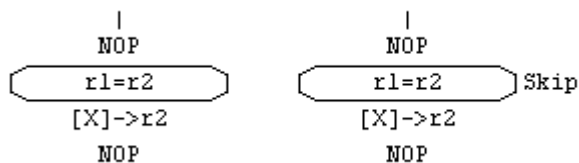
The operators for conditional skipping

The applied “CONDITION”-element must not have a vector or should have the reserved word “SKIP”.

Template	Comments	Example	Equivalent
R = R	Two working registers are equal	r0 = r1	CPSE R,R
R.# = 0	# bit in working register is equal to 0	XH.2 = 0	SBRC R,b
R.# = 1	# bit in working register is equal to 1	r14.3 = 1	SBRS R,b
P.# = 0	# bit in I/O register is equal to 0	PinC.6 = 0	SBIC P,b
P.# = 1	# bit in I/O register is equal to 1	EERE = 1	SBIS P,b

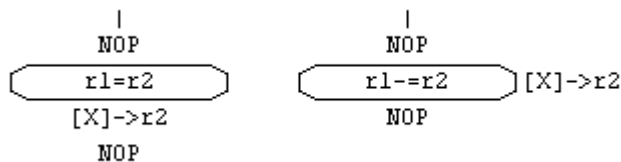
There are two ways to create the “CONDITION”-element for conditional skipping:

1. The vector and the vectors' name are missing or the reserved word “SKIP” is applied. In the following examples, the operation “[X]->r2” is skipped if the condition “r1=r2” is true:



2. The opposite condition is entered, the vector is missing, and the skipping operation is entered as a vector name. In this case, the compiler interprets this element as: if condition is true then execute operation.

The following examples are identical:



The comments for operators

The window with all existing operations for the selected microcontroller can be opened with the “View\Templates” menu item.

Note, some types of microcontrollers don't covers the complete set of templates of the above shown tables.

The “#” symbol in templates means a constant. The constants can be either positive or negative. For example, the operators:

`-2 -> r16` and `$FE -> r16`

are compiled equivalently.

Note to some peculiarities of creation the unconditional branches and subroutine calls:

- An unconditional branch can be created by means of the “JMP Vector” element only. If the length of the branch is not longer than 2047 items, the compiler is applied the short relative branch (RJMP), else the Long Branch (JMP k).

- A subroutine call can be created by the writing down a vertex (or label) name or a physical address in the “FIELD”-element. At that, if length of the branch is not longer than 2047 items, the compiler is applied the short relative subroutine call (RCALL), else the long subroutine call (CALL k).

Example:

	TestPins
...	...
TestPins	...
...	Ret

- An indirect branch or subroutine call (equivalent: “IJMP” and “ICALL”) can be created by means of using the “JMP[Z]” or “CALL[Z]” in the “FIELD”-element. Example:

	TestPins
...	...
TestPins -> Z	...
CALL[Z]	Ret
...	

The immediate configuration of constants

The “#” character in templates means a constant. It can be configured directly.

The Algorithm Builder supports the configuration of constants in hexadecimal-, decimal-, octal-, binary-, symbolic- as well as floating-point formats.

For the **decimal** format, the constant is configured as usual, by means of ciphers “0..9”. Example: “35” or “24000”.

For the **hexadecimal** format, the configuration starts with characters “\$”, “#H” or “#h” followed by characters “0..9”, “A..F” or “a..f”. Example: “#h0F”, “#He5”, “\$23E7”.

For the **octal** format, the configuration starts with the characters “#O” or “#o”, following by ciphers “0..7”. Example: “#o107”, “#O2071”.


For the **binary** format, the configuration starts with the characters “#B” or “#b”, following by character “0” or “1” are used. Example: “#b01101101”, “#B10011110”.

Signed constants are configured by a preceded positive or negative sign “+” or “-”, such as: “-54”, “+2”, “-\$5E3F”.

Double quotes define the **symbolic** configuration. In this context, the actual value of constant is an ANSI code of character. E.g., these configurations of constants: "0" and \$30 are equivalent. If a constant is composed by a string of characters, this constant is multi-byte. E.g. "12" is equivalent to \$3231.

The allocation of the microcontroller's resource and declaration of names

The development of programs with the Algorithm Builder is possible by using the standard names of the Working- and of I/O- registers. Additionally, the SRAM and EEPROM can be accessed directly by a physical address. However, it is very inconvenient.

The Algorithm Builder contains the specialized worksheet for the allocation of the microcontroller's resource and the declaration of names. To toggle editing of the algorithm and the worksheet, select the "View\Toggle Algorithm/Data table" menu item, press the "F12" key, or click the  button.

A name can be defined using the characters "A..Z", "a..z", ciphers "0..9", and underscores "_". The first symbol must be not a cipher.

The compiler is case insensitive therefore names such as "GenerateNoise", "GENERATENoise" or "generatenoise" are identical.

The worksheet contains six sections:

- A section for declaring working registers variables
- A section for declaring I/O registers names
- A section for declaring working registers and I/O registers bit names
- A section for declaring SRAM variables
- A section for declaring EEPROM variables
- A section for declaring constants

Data formats

The following data formats are supported:

Format	Key word
1 byte	"Int8" or "Byte"
2 bytes	"Int16" or "Word"
3 bytes	"Int24"
4 bytes	"Int32" or "DWord"
5 bytes	"Int40"
6 bytes	"Int48"
7 bytes	"Int56"
8 bytes	"Int64" or "QWord"

Note, the one-byte format is the default.

Section for declaring constants

The caption of the section is: **Constants:**

There are the following fields in this section:

- Name** – declaring name.
Value – immediate constant or algebraic expression.

The example:

Constants:		
Name	Value	Comment
Weight_Index	7	declaring the "Weight_Index" as a constant = 7
Weight_Min	10	
Weight_Typ	75	
Weight_Max	Weight_Min+100	
InputCode	\$DC5E	

To obtain some byte from a declared constant, it is required to insert ".#" to a name, in which the "#" is a byte number. E.g., the operator

"InputCode.1 -> r16"
stores \$DC in r16.

The following system constants are declared by default:

"CPU_Clock_Frequency" (Hz);
"IO_Org",
"SRAM_Size" (bytes);
"SRAM_Org" (bytes);
"EEPROM_Size" (bytes).
"Flash_Size" (16 bit words).
"Flash_Page_Size" – (16-bit words).

Section for declaring working register names

The caption of this section is: **Working registers:**

There are the following fields in this section:

Name - declaring name;
Index - (optional parameter) defines the register index. If this field is left empty then the next after the previous index is used;
Format - (optional parameter) [format](#) of the declared register. By default, the one-byte format is applied.

The example:

Working registers:			
Name	Index	Format	Comment
rr0	0	Word	two-byte register (r0,r1)
rr2		Word	two-byte register (r2,r3)
rrrr0	0	DWord	four-byte register (r0..r3)
Counter	16	Int16	two-byte register (r16,r17)
Ar			one-byte register (r18)

When a two-byte register is declared, the compiler automatically declares two one-byte registers composing it, with the characters "L" and "H" added to declared name. When a three-, four- or more-byte register is declared, the characters "0", "1" etc. are added. For example, when double register "Counter" is declared, the "CounterL" and "CounterH" names are be automatically declared.

Note, the multi-byte registers are used in macro-operators.

It is permitted to assign several names for one register.

By default, the standard names of one-byte registers are declared: "r0...r31" or "R0...R31", and: "WL"="r24", "WH"="r25", "XL"="r26", "XH"="r27", "YL"="r28", "YH"="r29", "ZL"="r30", "ZH"="r31" and for two-byte ("Int16" or "Word"): "W"=("r24:r25"), "X"=("r26:r27"), "Y"=("r28:r29") and "Z"=("r30:r31").

Section for declaring I/O register names

The caption of this section is: **I\O Registers:**

There are the following fields in the section:

Name - declaring name;
I/O Register - standard or already declared name of I/O register.

The example:

I/O registers:		
Name	I/O register	Commentary
InputPort	PortA	declaring "InputPort" as "PortA"
OutputPort	p#h22	declaring "OutputPort" as port # \$22

Different names can be assigned to the same register.

The standard names, such as "p0...p63" and "SPL", "SPH", "TCNT0", "EEDR" etc, defined in the original microcontroller guide are declared by default. Moreover, the double registers such as "ADC", "TCNT1" can be used in macro-operators.

Practically, it is no need to fill in this section inasmuch as all used I/O registers has a standard name by default.

Section for declaring bit names

The caption of this section is: **Bits:**

There are following fields in the section:

Name - declaring name;
Bit - bit definition.

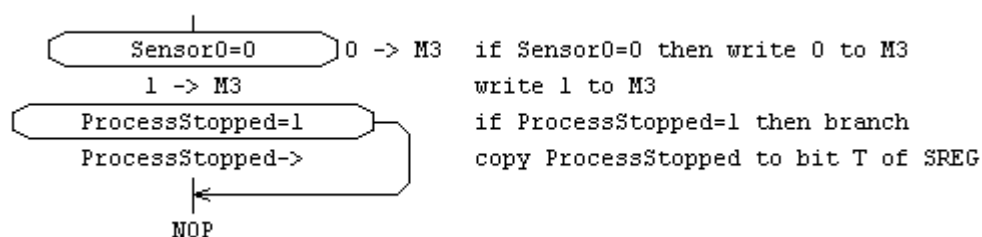
Example of then section filling:

Bits:		
Name	Bit	Commentary
Data	PortB.7	declaring "Data" as "PortB.7"
Clock	PortB.6	declaring "Clock" as "PortB.6"
OnChange	r0.4	declaring "OnChange" as "r0.4"

The standard names, such as "DDA7", "ACIC", "ADEN", "ICNC1" etc, defined in the original user guide of microcontroller are declared by default.

Declared bits can be assigned for Working- , I/O- register, as well as SRAM- or EEPROM- variables.

Example of the using of bit names:



Section for declaring SRAM variables

The caption of this section is: **SRAM:**

There are the following fields in the section:

Name - declared name;
Address - (optional parameter) physical address. If this field is left empty, it is used the next address after a previous variable or #60 (or #100) in the beginning of the compilation.

Format - (optional parameter) [format](#) of a variable. If this field is left empty, the one-byte format is used. The multi-byte formats are used with macro-operators.

Count - (optional parameter) number of declared variables (size of array). By default it is 1.

Example:

SRAM:				
Name	Address	Format	Count	Commentary
EditIndex				one-byte variable
ShowMode		Word		two-byte variable
Reg		Int24	4	four four-byte variables
LCD_Page			16	16 one-byte variables
Phase	\$100			one-byte variable at address \$100
XArray	@Phase+4		24	24 one-byte variables begins from address \$104

For the operators with direct SRAM addressing: “[#]->R” and “R->[#]”, a name of a variable can be used instead of “[#]”. A name with the “@” prefix is a constant which contains a physical address.

The following examples are compiled equivalently:

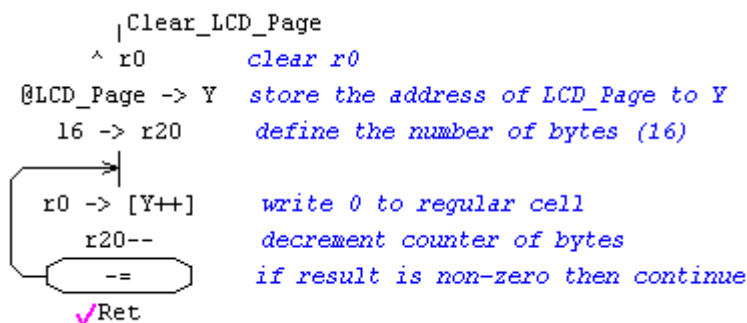
`[$100] -> r0 , Phase -> r0 , [@Phase] -> r0`

Multi-byte variables are used with macro-operators.

If a variable is declared as an array (Count>1, e.g.: “LCD_Page”), its name points to the first byte of an array. To address a random cell, use the offset. E.g. to copy r0 to the 5-th element of “LCD_Page” array, use the:

`r0->[@LCD_Page+5]`

In the following example all elements of the “LCD_Page” array are cleared:



Section for declaring EEPROM variables

The caption of this section is: **EEPROM:**

There are the following fields in the section:

Name - declaring name;

Address - (optional parameter) physical address. If this field is left empty, it is used a next address after a previous variable or zero in the beginning of compilation.

Format - (optional parameter) [format](#) of a variable. If this field is left empty, the one-byte format is used. Multi-byte formats are used with macro-operators.

Count - (optional parameter) number of declared variables (size of array). By default it is 1.

Value - (optional parameter) the initial value. It can be a constant (or algebraic expression of constants) or if Count>1, the typed with commas array of constants.

Alternatively, the initial values can be filled in by the “Load: FileName” directive, where the **FileName** is a name of the included file. See the [Direct inclusion of a data file](#)”.

Example:

EEPROM:					
Name	Address	Format	Count	Value	Comment
EE_Cell					one-byte variable
EE_LCD_Page			16	1,2,3,4,5,6,7,8	16 one-byte variables with initial values
InitValue		Word			two-byte variable
StartValue		Word	3	259,3331,0	3 two-byte variables with initial values
InitScale		DWord		#h27773F	four-byte variable with initial value
FileTable		Word		Load: Table.db	array of two-byte with filling from a file

The declared name can be used with macro-operators, working with EEPROM, e.g.:
 InitValue -> X, 1875 -> InitValue

The name with the “@” prefix is a constant which contains a physical address.

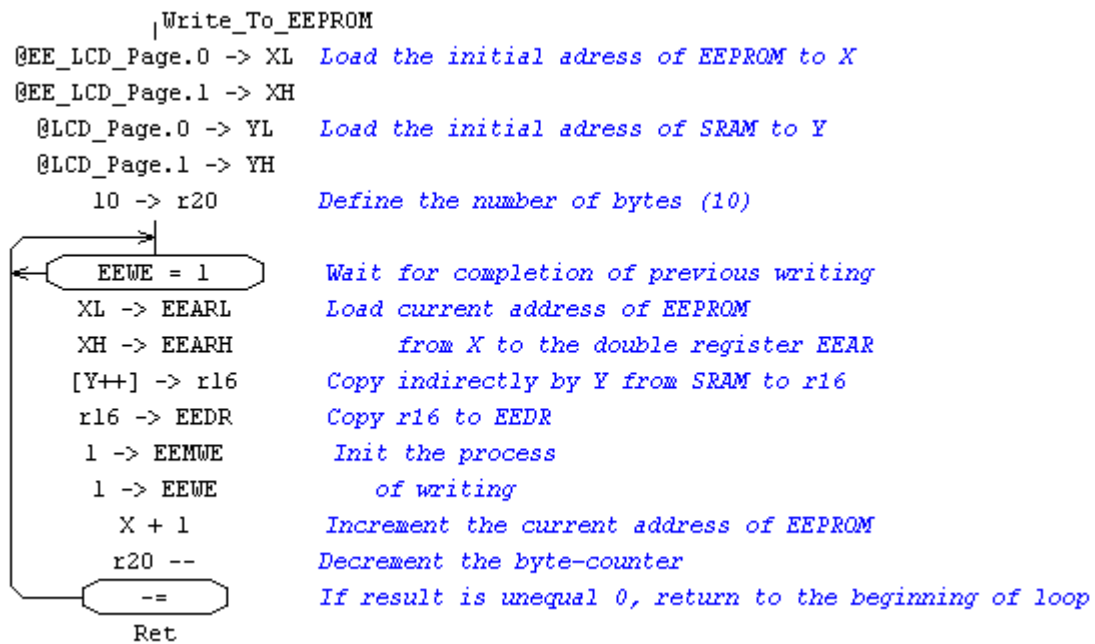
The example of copying the EEPROM array “EE_LCD_Page” to the SRAM array “LCD_Page”:

```

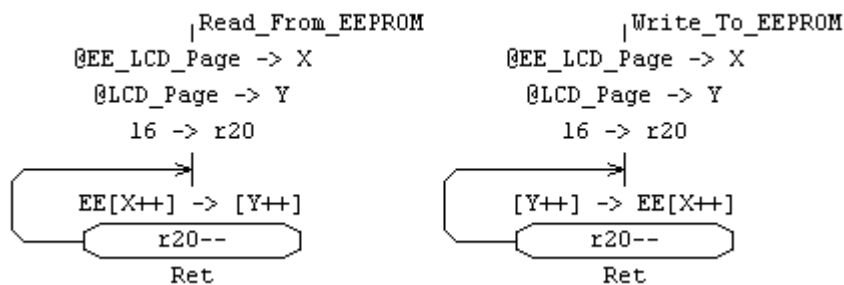
      |Read_From_EEPROM
@EE_LCD_Page.0 -> XL  Load the initial address of EEPROM to X
@EE_LCD_Page.1 -> XH
  @LCD_Page.0 -> YL  Load the initial address of SRAM to Y
  @LCD_Page.1 -> YH
    10 -> r20        Define the number of bytes (10)
  >
    XL -> EEARL      Load current address of EEPROM
    XH -> EEARH      from X to the double register EEAR
    1 -> EERE        Write 1 to EERE bitCopying
    EEDR -> r16      Copy EEDR to r16
    r16 -> [Y++]      Copy r16 indirectly by Y to SRAM
    X + 1            Incremen the current address of EEPROM
    r20 --           Decrement the byte-counter
    <- ->           If result is unequal 0, return to the beginning of loop
    Ret

```

The example of copying the SRAM array “LCD_Page” to the EEPROM array “EE_LCD_Page”:



With macro-operators these examples are compacter:



Representation of constants by algebraic expression

The denoted in templates as “#” constants can be defined with an algebraic expression.

The elements of such expression can be:

- immediate represented constants;
- names of constants, declared in the “**Constants:**” section of the table;
- LABELS (or VERTEXES) names, containing a physical program address;
- names of SRAM and EEPROM variables with the prefix “@”, containing a physical address;
- bit names with prefix “@”, containing their number in a register.

In expressions, it can be applied the following arithmetic operations:

- addition: “+”;
- subtraction: “-”;
- multiplication: “*”;
- division: “/”;

and the logic operations:

- AND: “&;”;
- “OR: “! ”;
- EXCLUSIVE OR: “^”.

For example, in the operator:

“[@LCD_Page+ 5*3]->r0” (template: “[#]->R”)

the “@LCD_Page+ 5*3” is a constant represented by an algebraic expression.

It is possible to create complicated expressions with round brackets.

Exponential functions are provided too. Syntax is "A(B)" where "A" is the basis and "B" the exponent. E.g., in the register "TIFR" of the microcontroller AT90S8515, the "OCF1A" bit is #6 and the "OCF1B" is bit #5. In this given case, the operations:

```
2 (@OCF1A)+2 (@OCF1B) ->r16 (template: "#->R")
r16->TIFR
```

are equivalent to:

```
2 (6)+2 (5) ->r16 or 64+32->r16 or #b01100000->r16
r16->TIFR
```

These operators clear the bits "OCF1A" and "OCF1B" of the register "TIFR".

Moreover, there are standard expressions, giving out properties of declared variables as a constant:

FormatOf (Var) - number of bytes of a "Var" variable (SRAM, EEPROM I/O or Working registers);

CountOf (Var) - number of reserved cells for a "Var" variable (SRAM or EEPROM) (a value of the "Count" field of the table).

E.g. in the above represented example of the table, it was declared the "Reg" variable with "Int24" format and count is 4.

For this variable, the "FormatOf (Reg)" expression will be equal to constant 3 and the "CountOf (Reg)" expression - to constant 4.

Macro-operators

Using only elementary operations for software development is inconvenient. E.g., to store the constant \$1234 to the double register X, it is necessary to store the high and low byte one by one:

```
$12->XH
$34->XL
```

It is insufficient visually. The need of possibility to write "\$1234->X" directly is evident. The compiler must separate out this operator to the two elementary operators: "\$12->XL" and "\$34->XH". When the constant is decimal, this possibility is needed even more.

Moreover, it is impossible to execute some operations directly by elementary operations. E.g.:

"37->r0" or "\$FF->DDRB".

In both cases, it is necessary to store a constant firstly in a working register (r16...r31) next, to copy it to r0 or DDRB.

The Algorithm Builder supports following macro-operation templates:

Arithmetical-logical macro-operators. The "FIELD" elements are using for these operations.

Template	Comment
* -> *	Copying
* + *	Arithmetic adding
* - *	Arithmetic subtraction
* & *	Bitwise logic AND
* ! *	Bitwise logic OR
* ^ *	Bitwise logic EXCLUSIVE OR
^ *	Clear (writing zero)
* ++	Increment
* --	Decrement
- * -	Bitwise inversion
- *	Arithmetic inversion
* >>	Logical right shift
> * >>	Logical right shift with carry
± * >>	Arithmetic right shift
<< *	Logical left shift
<< * <	Logical left shift with carry
* ->	Copying to stack (push)
-> *	Copying from stack (pop)
# -> * . #	Set or clear bit

0 -> ZH

Note, when using the signed values, the formats of both operands should be equal. The formats of operands in macro-conditions have to be equal too.

The operands with indirect addressing are one-byte only. For multi-byte operations with indirect addressing, it is necessary to add a colon and a format declaration as shown in the following example:

```
$AB3E->[Y]:Word.
```

Such operations are translated into the following elementary operations:

```
$3E -> r16
r16 -> [Y]
$AB -> r16
r16 -> [Y+1]
```

The execution of all multi-byte operations, except the right shifts, starts from a lowest byte. In case of right shifts, the action begins from a highest byte. Due to this fact, multi-byte macro-operations with indirect addressing have some limitations such as:

- It is impossible to use the operands with the pre-decrement in the operations, starting from the low byte, e.g.: `--X]:Word + 24000;`
- It is impossible to use the operands with post-increment in the operations, starting from the highest byte, for example: `<<[Z++]:Int24;`

Also, it is impossible to create the multi-byte macro-operation with indirect addressing using the operand `[X]`, because in AVR the indirect addressing operation with offset (`[X+#]`) does not exist. Use `[X++]` or `--X]` instead.

If the creation of any macro-operation is impossible, the compiler will generate a warning message.

Note, when writing to the EEPROM, the compiler includes the code of waiting the end of process. The execution of this code requires some milliseconds. To define the sequence of writing to EEPROM operators, select the "Options\Project options" menu item and choose the "EE Macro" tab.

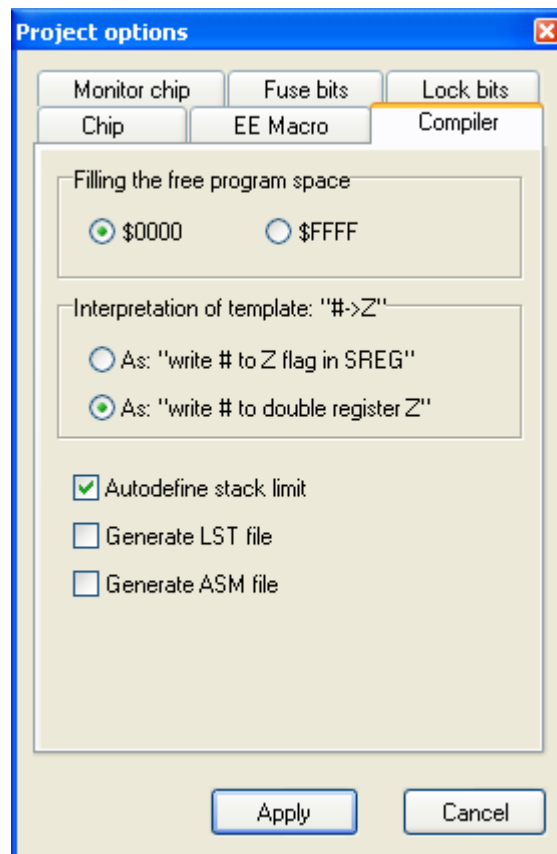
The compiler translates the macro-operators to a set of elementary microcontroller instructions. The working registers `r16` and `r17` are used in most of these operations. Therefore, in order to avoid errors, it is recommended not to use these registers and it is indispensable to save these registers onto the stack when calling interruption-handling subroutines.

When using the macro-operators with working registers and constants, it is preferably to use working registers of the upper half (`r18..r31`). Else, extra operations loading a constant to the `r16` are used. It increases the size of the code, but it not will cause an error.

Note, the activated by macro-operations status flags in the `SREG` will not behave normally and are unreliable to use.

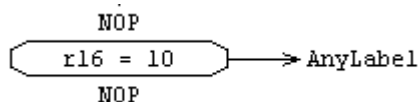
For the development of cycles, it is convenient to use the macro-condition "`* --`". It decrements an operand and branches out if the result is non-zero.

Note: the "`#->Z`" template can be interpreted in two different ways. As: "write `#` to the `Z` flag in the `SREG`" or as: "write the `#` to the double register `Z`". To solve this dilemma, go to the project options:

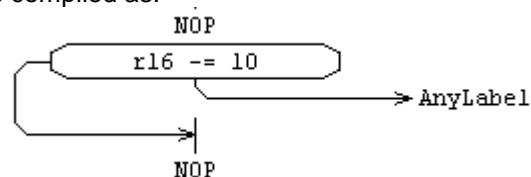


But, when an operator is selected by bold font, by pressing of the "F2" key, it will be interpreted only as: "write the # to the double register z» and the "#-> .z" template is interpreted exclusively as: "write # to the z flag in the SREG" independent of option.

In AVR the length of conditional branching is limited by ± 63 program items. However, if the applied length exceeds this limit, the compiler uses more complex construction with using of opposite condition and unconditional branching. E.g., in this case the conditional branching:

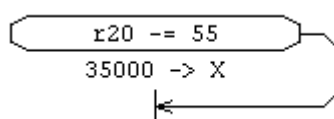


is compiled as:



Conditional operations

Depending on some condition, it is often required to perform only one operation, e.g.:



In this example if `r20=55` then `35000` is stored in `x`. In such cases, it is more convenient to carry out these actions by *conditional operations*. By doing so, the condition in the contour must be inverted, the operation must be stored as vector name and the vector itself must be absent:

`r20 = 55` `35000 -> X`

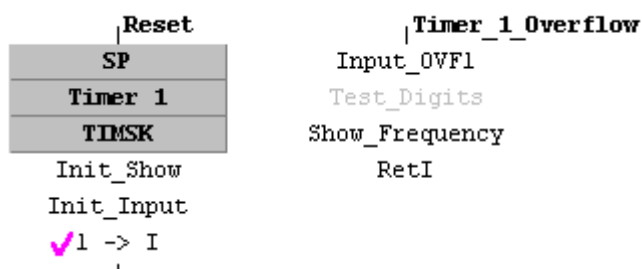
Labels for interrupt handling

For handling of interrupt, it is necessary to place an unconditional branch to a appropriate subprogram at its required program address. Using special *labels for interrupt handling*, the compiler can able configure the interrupt vectors automatically. This can be performed by assigning to a label a standard interrupt name and marking it as a macro-operator by the “F2” key. An easier way is to select the “Object\Interrupts vector...” menu item.

Having found at least one such label, the compiler fills the space of interruption vectors with the “RETI” code and at the appropriate to a interruption address puts the code of unconditional branch to a given label.

Note, if the interrupt handling labels are in use, the initial program addresses are filled in with the “RETI” code or an unconditional branch code. Therefore, the regular start of the program execution must begin from the “Reset” label.

Example:



Accordingly, to create an interruption, you should provide the following:

1. Enter the “Reset” vertex.
2. Set the stack pointer with “SP” SETTER. (Usually, it is an upper SRAM address).
3. Enable this interruption. (For the Timer/Counters it is a appropriate bit of the “TIMSK” I/O register).
4. Enable the global interruption with the “1 -> I” operation.
5. Enter an interrupt handling subroutine, which should be started from the appropriate named vertex, and terminate with the “RetI” operator.

To handle the interrupt in the BOOT section, use names with the “BOOT_” prefix.

Immediate data allocation into the program memory

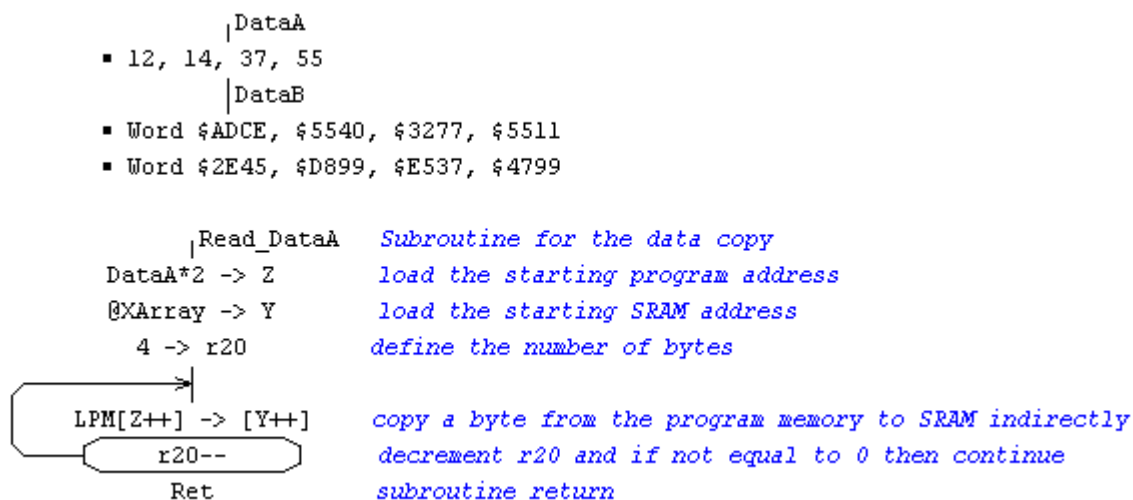
Additionally, to the algorithm, the random data can be placed into the program memory, such as code tables or text messages. The data can be placed after the dead-end operators such as “RET”, “RETI”, or an unconditional branch. The “FIELD” element is used for this. To switch this element to the immediate data mode, use the “Shift+F3” keys or select the “Element\Data” menu item. In this mode the string of element will be left aligned and at left side will appear a little black square.

For the following definition of their program memory address, it is required to place a label before these.
Syntax:

`[Format] #, #, #..`

The format (data size) is an optional parameter. By default, the one-byte data size is applied.

Example:



Note, when data reading with "LPM" operation, it is required to store a **doubled** address in the register **Z** inasmuch as this operation works with the byte-serial program memory.

Note, that the compiler rounds up the number of bytes in the data line. Thus, if the number of bytes is not even, the last byte is filled in with zero.

The data may be placed as a text. But in this case, the format must be only one-byte. The text must be in the double quotes as shown in the example:

```
"Hello! "
```

For the text data, the ANSI codes of characters are stored in the program memory.

Sometimes, the developing equipment works with character codes which differ from the ANSI. In this case, using the appropriate recoding file, the compiler can modify the code of characters automatically. To recode, add in the round brackets after the right double quote the name of this file. For example:

```
"ПРОЦЕСС"(LCD_CYR)
```

Such files must have the ".dcd" extension. The package of the Algorithm Builder contains the "LCD_CYR.dcd" file, which adapts the character codes to codes of an alphanumeric liquid crystal display with Cyrillic. By analogy with this file, you can create other files.

A mixed data representation is allowed:

```
"HELLO! ", $0, $DB
```

Direct inclusion of a data file

Algorithm Builder supports the direct inclusion of a data file into the program memory or EEPROM. To do this, use the directive: "Load: FileName", where **FileName** is the name of the included file. The file must have one of the four following formats:

- IntelHEX (the ".hex" extension);
- Generic (the ".gen" extension);
- Binary (the ".bin" extension);
- Algorithm Builder data format (any other extension), which is a textual file with data as in the described above "immediate data allocation" chapter.


Example:

```
$11, $22, $33
$44, $55, $66, $77, "HELLO!"
Word $ABDC, $FEDC
```

Inclusion of other files with algorithm in the project

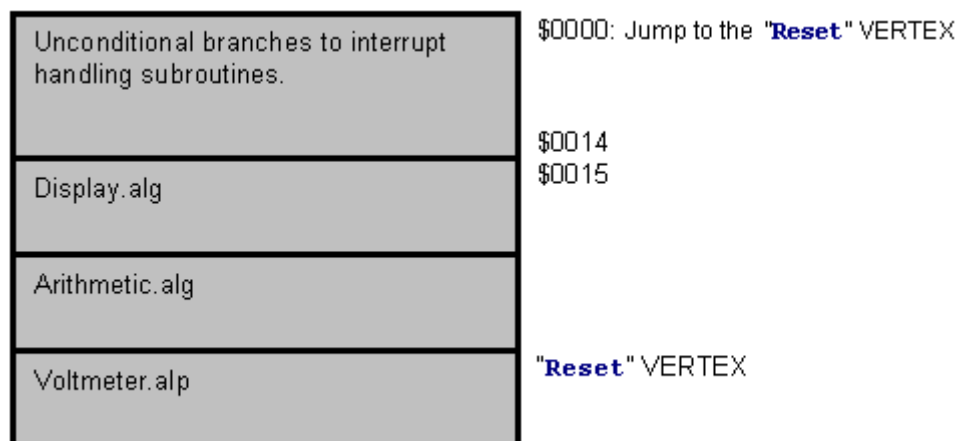
When a new project is created, a leading file with ".alp" extension is produced. This file can be unique. But, it is more convenient to divide the project into several files with functional fragments. Moreover, it allows using already developed algorithms.

To add a new file, select the "File\New" menu item.

The Algorithm Builder requires the location of all project files in the one common directory with the leading file ("*.alp"). To open a file, select the "File\Open..." menu item or click the  button.

To add a file to a project, use the directive: "Include: FileName" or "+: FileName", where FileName is the name of the including file. Use the "TEXT" or "FIELD" elements for this directive. Having found this directive, the compiler temporally suspends the work with the current file and goes to a mentioned file.

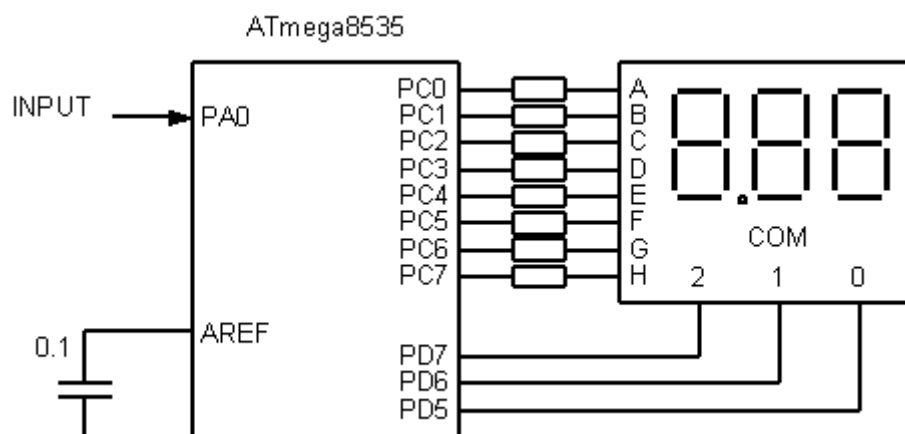
In the following example of a voltmeter, the algorithm is separated in three files: "Voltmeter.alp", "Display.alg", and "Arithmetic.alg". In accordance with locations of "Include:" directives, the program is arranged in the following way:



Manipulating of directives location, you can randomly arrange the program memory.

The example of a voltmeter

This example contains more complete possibilities of software development with the Algorithm Builder.



The voltmeter measures voltages in the range of 0 to 2 volts with the resolution of 10 mV and shows the value at the 3x7-segment LED display.

The algorithm is placed in the "EXAMPLES\VOLTMETER" directory.

The project contains three files: "Voltmeter.alp" as a main project file along with "Display.alg" and "Arithmetic.alg".

The CPU clock source is defined by default with the internal RC generator 1 MHz (Fuse bits: CKSEL = 0001).

The execution of the algorithm starts with the jumping to the "Reset" label at the "Voltmeter" page. Next, to support the subroutine calls, the "SP" setter stores the end address of the SRAM (\$25F) in the stack pointer. The "Timer 0" setter assigns the clock source of Timer/Counter0 as "CK/8", that provides the overflow period of 2.048 ms. The "TIMSK" setter enables the interruption on the Timer/Counter0 overflow.

The "ADC" setter assigns the prescaler to "CK/8" that provides the ADC clock frequency of 125 kHz (must be between 50 and 200), assigns the input multiplexer to ADC0 (PA0 pin), selects the internal voltage reference of internal 2.56 V, enables the interruption and sets the cyclic start by means of the Timer/Counter0 overflow.

Next, it is called the "Init_Display" subroutine (the "Display" page).

This subroutine sets the Port C and 5,6 and 7 bits of the Port D to the output mode, resets the index of displaying segment ("DigitIndex" variable), and clears the values of displayed digits (three byte "Digits" array).

The "1 -> I" operator enables the global interruption next, the execution is looped. Further, the algorithm executes on interrupts only.

When the Timer/Counter0 overflow occurs (with period of 2.048 ms), the "Timer_0_Overflow" subroutine is called by hardware. There is an array of 7-segments codes of digits from 0 to 9 at the "DigitCodes" label in the flash memory. Every 2.048 ms, the "ShowNextDigit" subroutine changes an index of activated digit (the "DigitIndex" variable) between 0 and 2, takes out the next digit from the "Digits" array, converts it to the 7-segments code and stores this code in Port C. At the same time, it changes the corresponding active cathode of the display by means of shifting the zero in-between PD5-PD7. Thus, about every 2 ms, the display shows a new digit.

On one's completion of conversion, the ADC generates an interrupt to call the "ADC_Complete" subroutine. This subroutine multiplies a conversion result by "k_ADC" coefficient by means of the "fMul_XY" subroutine in the "Arithmetic" page. This subroutine multiplies the X and Y double registers and stores a result in the Z register by means of the formula:

$$X * Y / 65536 \rightarrow Z.$$

This multiplication transforms a conversion result to a tens of millivolts. The calculation of the "k_ADC" coefficient is in the table of the "Voltmeter" page. Beforehand, it is required to measure the internal voltage reference by means of an external voltmeter at the "AREF" pin and to enter this value as millivolts (the "Vref" constant).

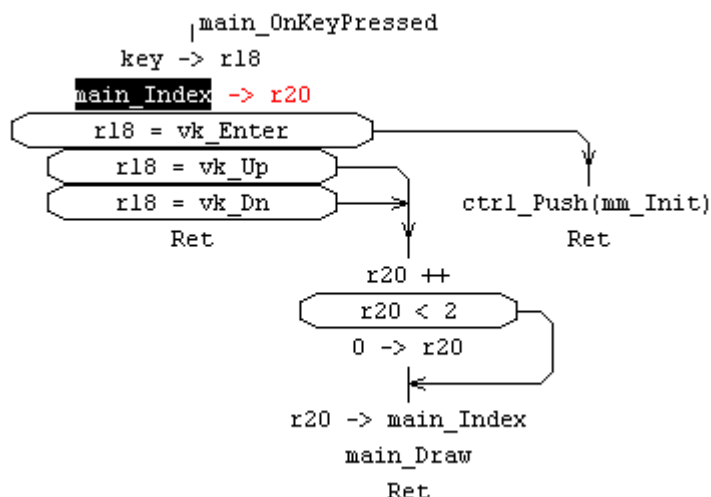
Next, the "Z_to_Digits" subroutine extracts from a value the hundreds, tens and ones next, stores these digits in corresponding cells of the "Digits" array.

Editing the algorithm

To select an editing element, use the "Up" or "Down" keys or click on an element with the right mouse button. When using the "Up" or "Down" keys, the elements are alternating according to their location in the memory.

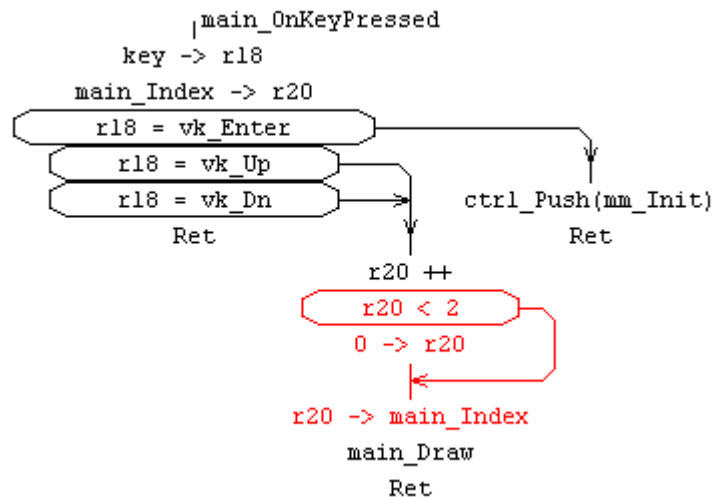
The selection of a fragment. There are three situations:

1. The selection of an in-string fragment.



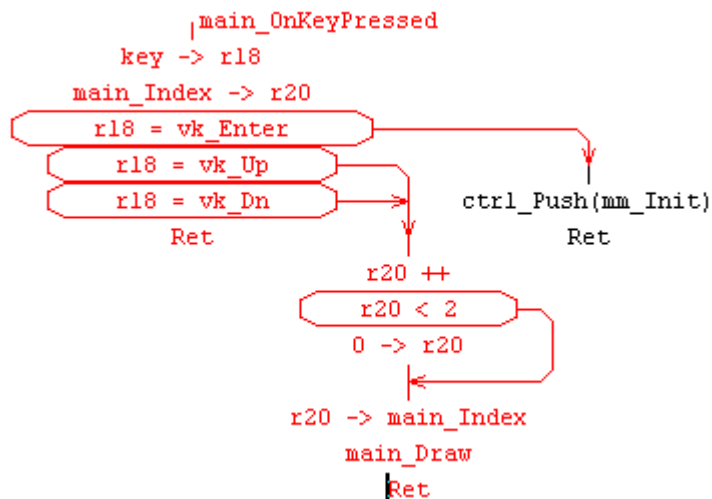
Use the "Shift+Left" and "Shift+Right" keys.

2. The selection of an in-block fragment.



Use the “Shift+Up” and “Shift+Down” keys. In this case, a “VERTEX” element *should not be* among the selected elements.

3. The selection of a complete block or several blocks.



Use the “Shift+Up” and “Shift+Down” keys. But in this case, at least one “VERTEX” element *should be* among the selected elements. Moreover, the selection of blocks is possible by means of pressing the “Shift” key while handing a window by clicking and dragging a left mouse button. Also to select block, click the left mouse button on it with the “Ctrl” key.

By default, the editing element is regarded as selected.

To **copy** a selected fragment or an editing element in the buffer, use the “Ctrl+C” or “Ctrl+Insert” keys.

To **delete** without the copying in the buffer, use “Ctrl+Delete” keys and with the copying – “Ctrl+X” or “Shift+Delete” keys.

To **paste** from the buffer, use the “Ctrl+V” or “Shift+Insert” keys.

There are three variants:

1. If the buffer contains an in-string fragment, the pasting will be into the string to the right of the cursor.
2. If the buffer contains an in-block fragment, the pasting will be in the block underneath the editing element.
3. If the buffer contains a complete block or several blocks, the first, the gray contour of the inserting fragment appears next, it is necessary to select the position of pasting with the mouse or with the “Up”, “Down”, “Left” or “Right” keys. To finish the pasting, click the left mouse button or the “Enter” key. To cancel the pasting, press the “Escape” key.

The “F5” key sets and clears a **breakpoint**. Breakpoints are shown by the  mark.

The “F2” key switches an element to the **macro-operator state**. Elements in the macro-operator state are shown by a bold font.

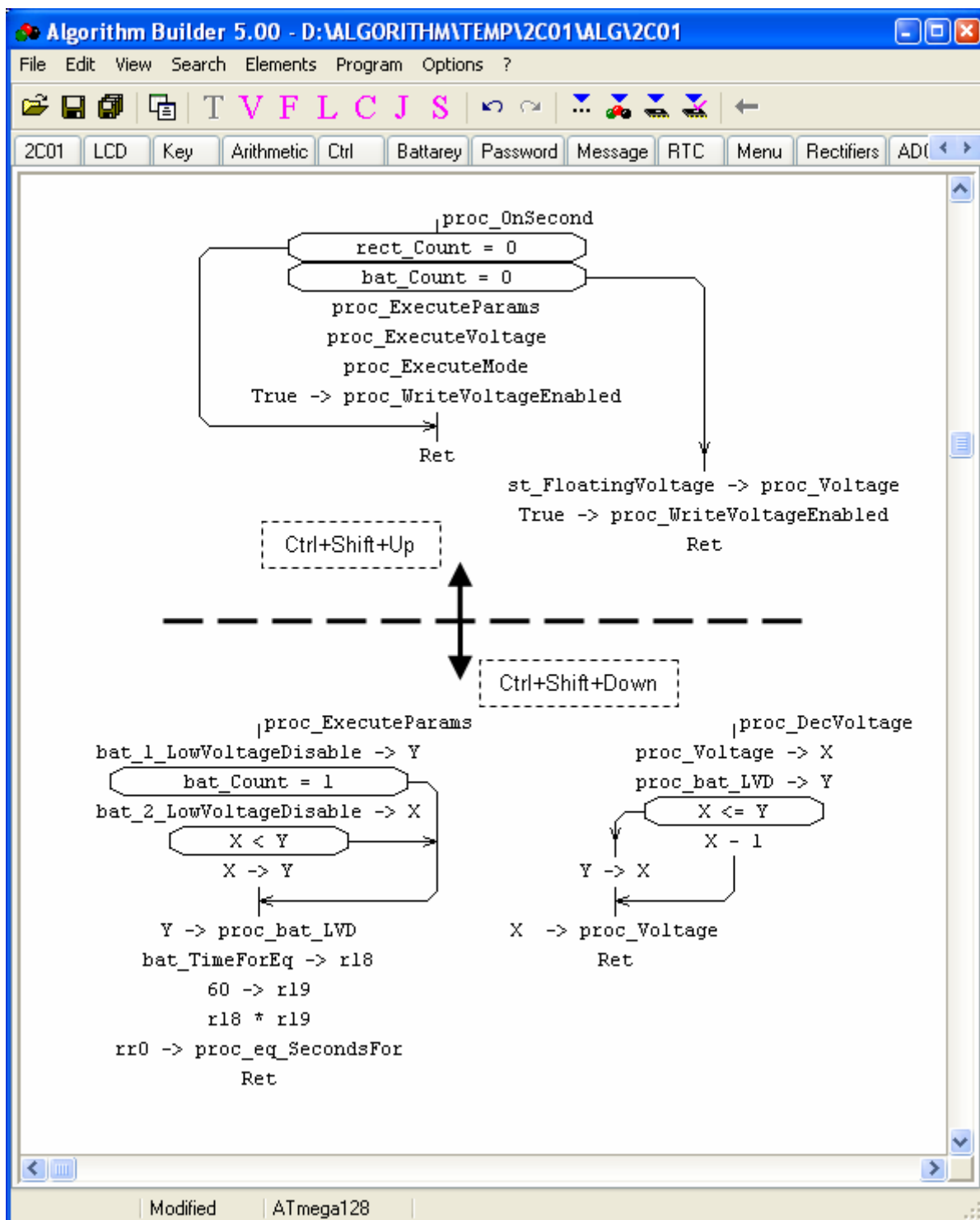
To **scroll** the working space, use appropriate scroll bars or the mouse by having pressed the left button on free space.

The “PgUp” and “PgDn” keys move the editor focus to the **next or previous** block vertex.


The “Shift+PgUp” key goes to the **beginning of the algorithm**, and “Shift+PgDn” – to the **end**.

To **move a block**, select any item in it next, use the “Ctrl+Up”, “Ctrl+Down”, “Ctrl+Left”, or “Ctrl+Right” keys, or drag it by the left mouse button.


The “Ctrl+Shift+Up” or “Ctrl+Shift+Down” keys **move the selected block and all following blocks**. This feature can be used to arrange an vertical free space in the algorithm or to delete a needless free space:



To **move a group of selected blocks**, use the left mouse button or the “Up”, “Down”, “Left”, or “Right” keys. To cancel the selection and to finish the moving, press the “Escape” key.

To **undo** the prior manual actions, use the “Alt+BkSpace” keys or the  button.

To **put the blocks in a required order** for the purpose of optimization the program, the first, select the several blocks. It is preferably to carry out it by means of the left mouse button with the “**Ctrl**” key in the required order next, select the “**Edit\Group selected blocks**” menu item.

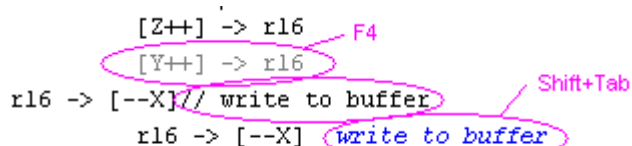
To jump to a vertex (or a label), double click on their name (jump to a subroutine body). To return, press the  button or select the “**Edit\Back**” menu item.

Comments.

The comment can be created by a few ways:

1. To disable the editing element, press the “**F4**” key. Such element is displayed by silver color (by default) and will be ignored by the compiler.
2. The compiler ignores the fragment of a string which follows the “**//**” characters.
3. To create an individual comment, use the “**Shift+Tab**” keys.

```
[Z++] -> r16
[Y++] -> r16
r16 -> [--X]// write to buffer
r16 -> [--X] write to buffer
```



Editing the resource allocation table

The editor focus can be moved by the “**Up**”, “**Down**”, “**Left**”, or “**Right**” keys. The “**Tab**” key moves the editor focus to the next field and the “**Shift+Tab**” – to the previous field. Also you can set the editor focus by pressing the left mouse button on a desired cell.

To **insert a new** line below the current one, press the “**Enter**” key, and to insert above the current one, press the “**Insert**” key.

To **select** lines, use the “**Shift+Up**” and “**Shift+Down**” keys. The current line is selected by default.


To **delete** a current line or selected lines without copying in the buffer, use the “**Ctrl+Del**” keys; to delete with copying in the buffer, use the “**Shift+Del**” or “**Ctrl+X**” keys.

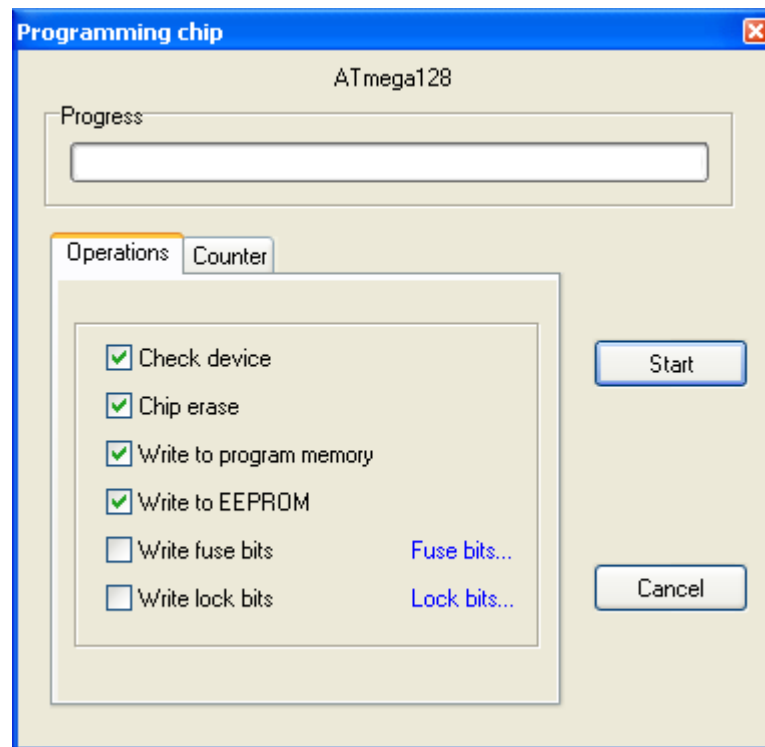
To **copy** a current line or selected lines in the buffer, use the “**Ctrl+Insert**” or “**Ctrl+C**” keys.

To **paste** copied lines from the buffer, use the “**Shift+Insert**” or “**Ctrl+V**” keys. Note, the pasting is possible *only in the current section* or in the same section of an other pages.

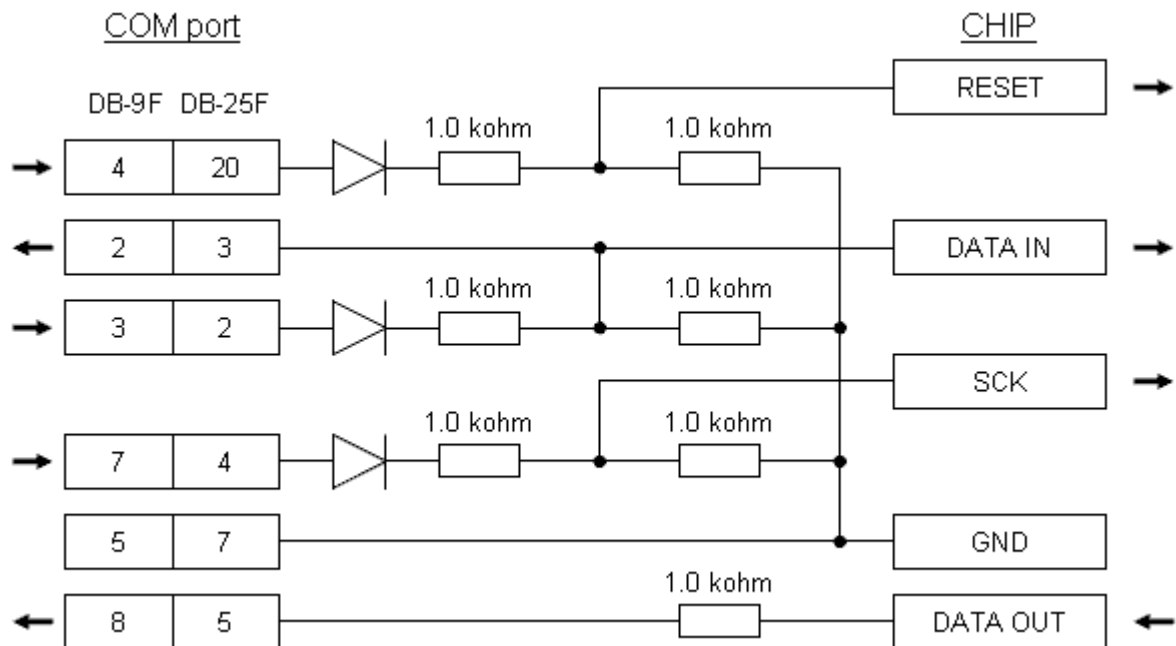
To change a **column width**, move the mouse cursor to its border in the caption line, and drag using the left mouse button.

Programming the microcontroller

The Algorithm Builder contains an in-build in-circuit programmer, which provides serial programming of chips. Choosing the “**Program\Run with Chip**” menu item, pressing the “**Ctrl+Shift+F9**” keys, or clicking the  button on the tool bar starts the compiler next, in case of absence of errors the following programmer window opens:



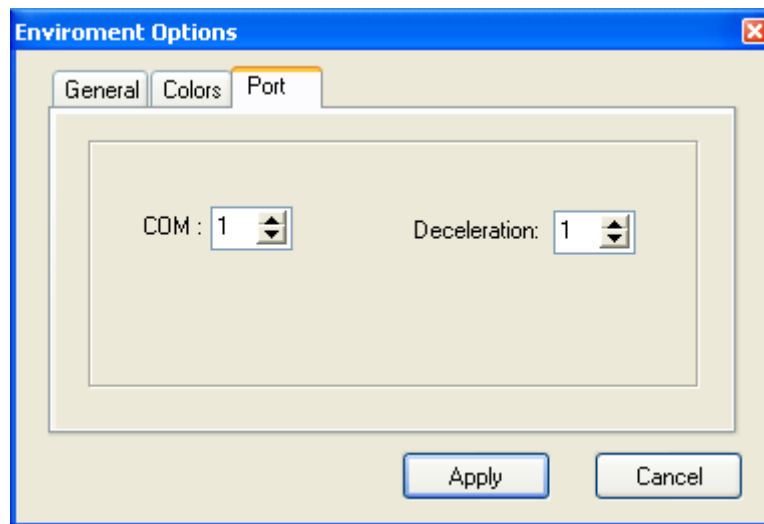
The microcontroller can be connected to a COM port by means of the following simple adapter:



The wattage of resistors is 0.125 W; the diodes are any type of impulse diodes with recovery time less than 50 ns (for example, the 1N4148).

Note, this adapter is adapted to the standard 12-volt COM port. However, on some computers contains the 6-volts COM port. In this case, the nominal of the connected with diodes resistors must be decreased to 100 Ohms.

The required COM port can be chosen in the environment options window ("Options\Environment Options" menu item):



The length of a cable connecting the COM port and a microcontroller should not exceed one meter (40 inches). It is recommended to use the ribbon cable, in which the signal lines should alternate with ground (GND).

Note the state ☒ of a Fuse bit means “unprogrammed”.

The external circuits of a design must not interfere with signals of the computer. When programming the microcontroller, the CPU clock frequency must be not less than 1MHz.

When the programming is finished, the “RESET” signal toggles from 0 to 1, thus, the program execution is started.


For electrostatic safety, it is necessary to connect the ground circuit at first.


Lock and Fuse bits.


The modification of lock and fuse bits is in the project options window ("Options | Project options..." menu item). In this window, it is possible to read these bits from a connected chip and to write independently of the programmer. However, if the "Lock bits" or "Fuse bits" options are checked in the Programmer window, these bits will be programmed during the general programming.

Attention! Programming of some Fuse bits might lock the further serial programming (such as “RSTDSIBL”, “CKSEL” etc.).



Debugging the algorithm in the simulator


To run an algorithm in the simulator, select the “Program\Run with simulator” menu item, press the “F9” key, or click the  button in the tool bar. At first the algorithm will be compiled.

For the step-by-step execution with call to subprograms and macro-operators (Trace into), use either the “F7” key or the  button in the tool bar.


For the step-by-step execution without call to subprograms and macro-operators (Trace over), use either the “F8” key or the  button in the tool bar.


For the execution up to an exit from a current subprogram, use either the “F6” key or the  button.

To start the continuous execution up to breakpoint (), press either the “F9” key or  button.

To start the continuous execution up to selected element, press either the “F4” key or  button.

In both cases, the execution can be interrupted by pressing of the “F2” key or  button.

To set or delete a breakpoint () at a selected element, press the “F5” key.

When the execution of the algorithm is stopped, the  mark points to a next unexecuted operator.

The current state of various components of the microcontroller may be observed by opening the appropriate windows by means of the “View\...” menu items. To control these windows, use the popup menu, having pressed the right mouse button.

To add a new watch in the “Watch” windows, click the “Add watch” popup menu item. To select the placed items, use the left mouse button with the “Shift” or “Ctrl” key. To move a selected item upwards or downwards, use the “Up” or “Down” keys, but in this case, only one item must be selected.

Also, you can watch a value of a variable directly in the algorithm by pop up hint at its name. To edit, double click on it.

In tabular windows (Watches, Maps), if an executed operation stored in some cell, this cell is highlighted by magenta color and if this cell was modified, then it is highlighted by red color.

To observe the duration of process, use the "Process time" window. It contains four independent microcontroller cycle counters. When the "Enable" flag is checked, counter can stop execution after the adjusted number of cycles. Setting the "Clear after stop" flag automatically clears the counter. When a counter stops the execution, "STOP" is displayed in its window.

I/O points.

There are two directives for input (output) current values of any variables from (into) a file.

Syntax:

File:FileName -> * (for input)

and

* -> **File:FileName** (for output)

where: "*" – any variable name (working, I/O register, SRAM or EEPROM variable);

"FileName" – name of a file

For example, the directive "**File:Data.abd -> X**" loads a current value from the "**Data.abd**" file to the double register "**X**".

The format of the file is defined by an extension and it can be one of fourth described in the "[Direct inclusion of a data file](#)" section.

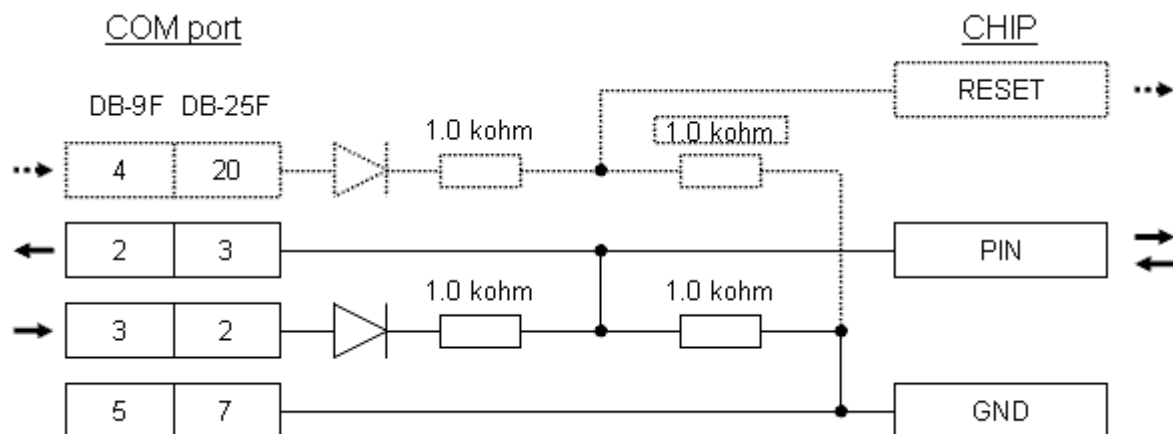
Note, that these directives are not in the program. They are only simulator actions and they are active only in the simulator. The size of a file should not exceed 256 K bytes. The output file is created on the disc after the closing of the simulator.

On-chip debug

For on-chip debug, the compiler adds to the program a small (160 words) hidden fragment. This fragment provides the transferring of all internal state of a microcontroller to the computer for displaying in the appropriate windows. At the same time, a value of any register, SRAM or EEPROM variable can be modified.

The connection of the microcontroller and the computer uses only one pin-out, which can be defined by the user. But, you may use the additional wire in order to reset the microcontroller.

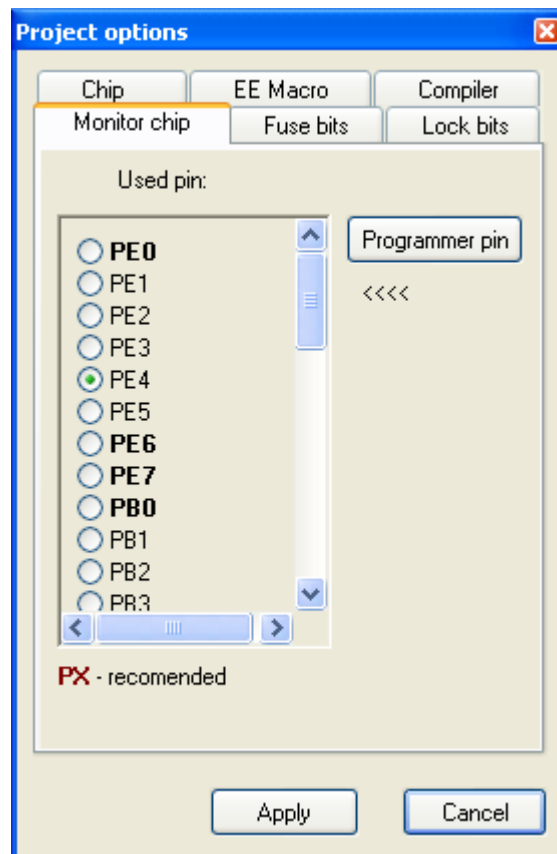
The recommended scheme of the adapter for on-chip debug:



You can use the adapter for the serial programming, but in this case the used pin is "DATA IN" only.

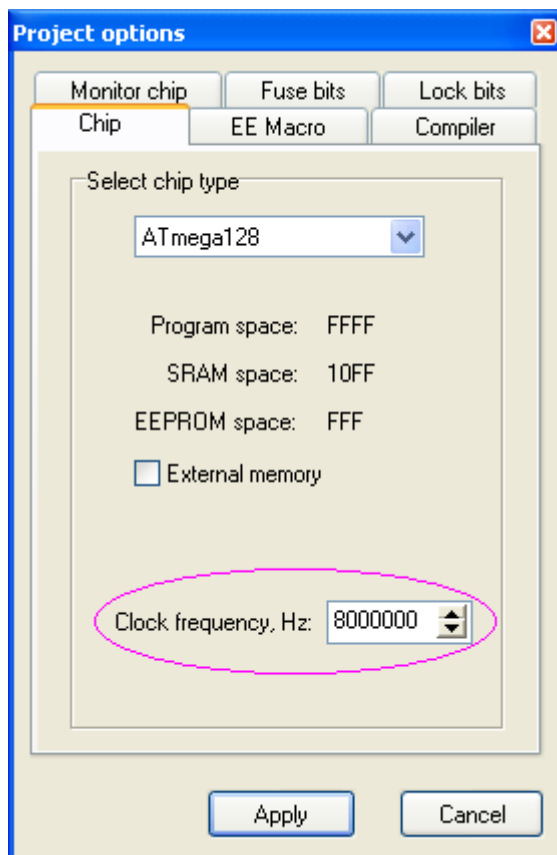
For work with on-chip debug you have to provide the following:

1. Define a used pin of the microcontroller. To do this, open the project options window at "Monitor chip" tab ("Options / Project options" menu item):



The recommended pins are marked by bold font and brown color. They don't have an alternative to the port output functions.


2. Enter the CPU clock frequency exactly:



3. Before a first breakpoint, the stack must be defined.

It is necessary to mean the following:

1. The including additional code increases the program size by 136 words + 1 or 2 words for each breakpoint (code of the subroutine call).
2. Debugger requires 11 bytes of the stack.
3. At a breakpoint, the program execution is stopped and the global interruption is disabled.
4. The on-chip debug is impossible for chips without the SRAM.
5. A using pin shouldn't be busy by alternative output signal (such as USART, SPI etc).
6. The program with on-chip debug code is unfit for normal use. Consequently, after the debugging, a chip must be reprogrammed in the normal mode.

To start the program execution with on-chip debug, click on the  button in the toolbar. After compilation, the programmer window opens. Start the flash memory programming by pressing the "START" button. When the program execution gets to a breakpoint, the windows with microcontroller's state are activated.

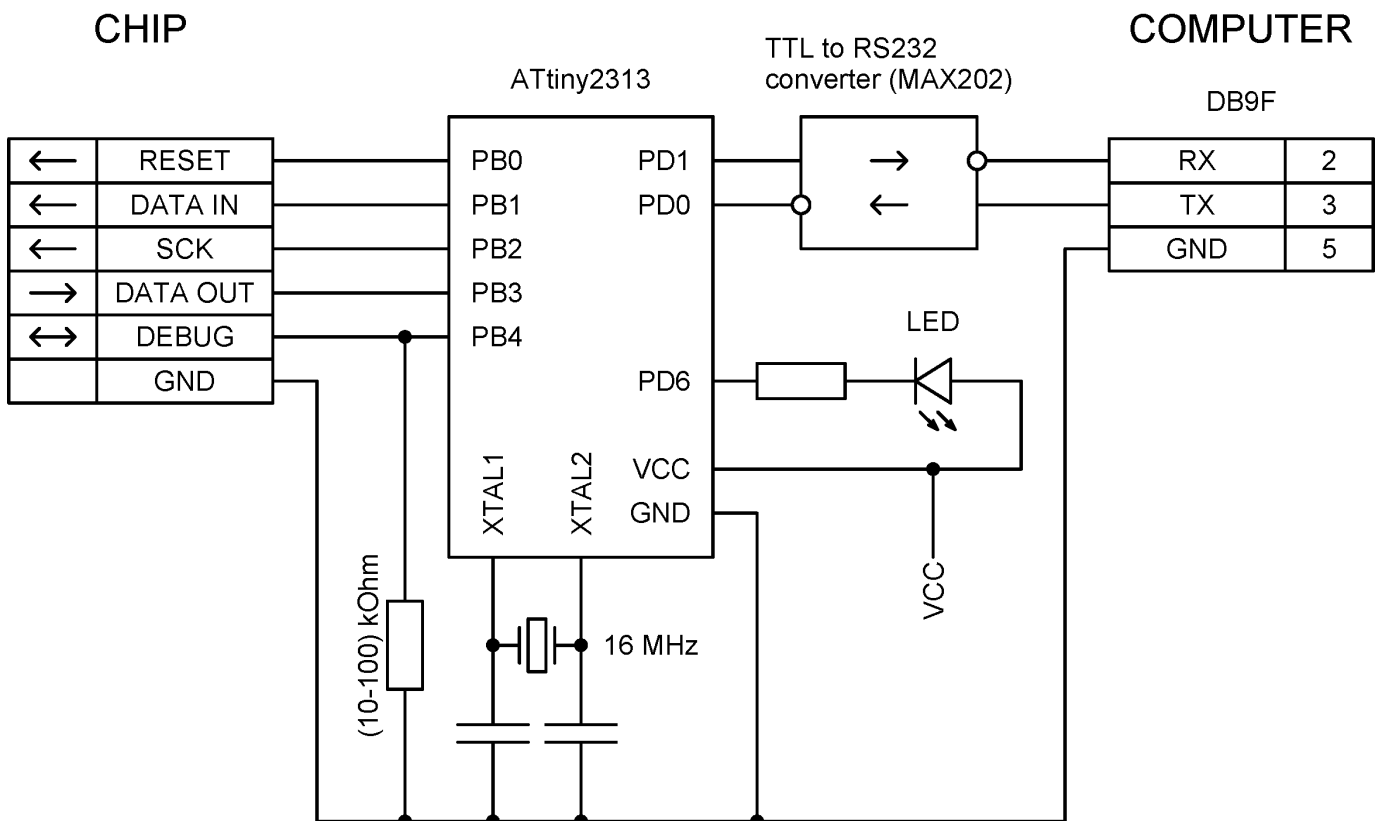
Note the data communication between a computer and a chip uses the asynchronous serial transfer. It requires the exact frequency concordance. Therefore, it is recommended to employ the on-chip debug with using the crystal resonator. Moreover, the CPU clock frequency must be entered with taking into consideration of the programmed frequency variation (for example, by XDIV or OSCCAL I/O registers).

The active adapter for programming and on-chip debugging

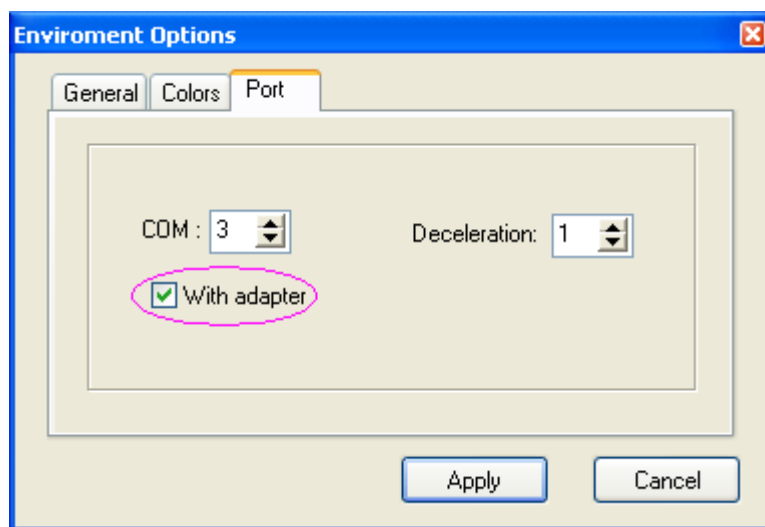
The represented above simple adapter provides the satisfactory work with a standard COM port. However, the speed of programming is too limited. When the project is small, this is not so important inasmuch as the programming lasts only a few seconds. But when the project exceeds 10 kB, the process can require a few minutes.

Furthermore, many of computers don't have a COM port and the using of a USB-RS232 converter is impossible for such connection.

The represented below active adapter provides programming with maximum speed and allows using the USB-RS232 converter.



The program for ATtiny2313 is placed in the "EXAMPLES\CommAdapter" directory. The chip can be programmed by the simple adapter or by any other programmer. To use this adapter it is necessary to set appropriate flag in the environment options:



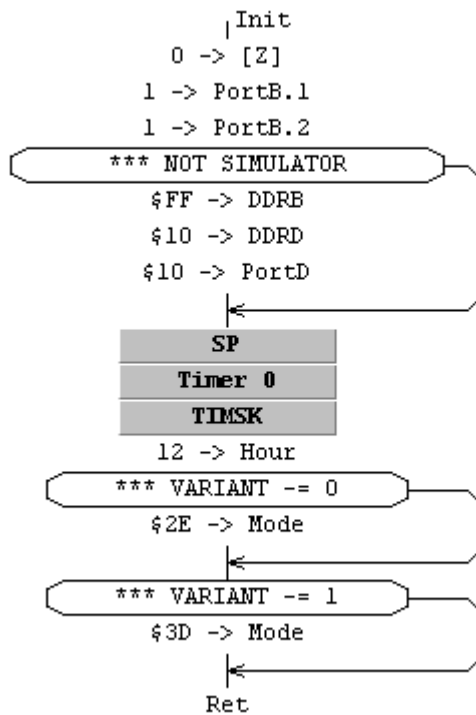
At low CPU clock frequency the programmer speed can be too high. In this case try to increase the deceleration. By default, most microcontrollers work with the internal 1MHz RC clock generator. This frequency requires the deceleration of 2.

The variant of the scheme with an **optical isolator** is in the file: "EXAMPLES\CommAdapter\OpticalIsolator.pdf".

The conditional compilation

The compiler allows including some algorithm fragments into the program on various conditions. For example, sometimes, when debugging an algorithm, it is required to include some fragments only for debugging. Sometimes, there is also the necessity of selective inclusion of fragments depending on values of some declared constants. It allows using one project file for a various variants.

The conditional compilation can be created using the "CONDITION" element. The compilation condition must start from three stars: "***". The condition can be one of the reserved words: "Simulator" or "Not simulator", or an algebraic comparison operation with constants. For example:



In this example the fragment:

```

$FF -> DDRB
$10 -> DDRD
$10 -> PortD

```

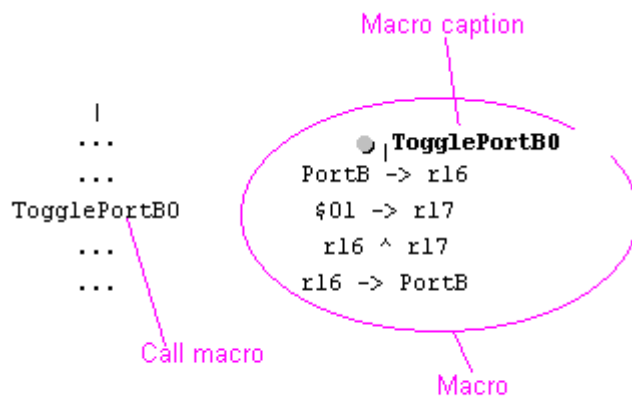
is compiled only for simulator;

the fragment: “\$2E -> Mode” is compiled if the value of the declared “Variant” constant is 0, and the fragment: “\$3D -> Mode” is compiled if the value of the declared “Variant” constant is 1.

User macros

The Algorithm Builder allows the creation of user macros. The macro body can occupy only one block of operators. The macro caption is a string of this block vertex. To switch a vertex to the macro vertex status, press the “Shift+F2” keys or select the “Elements/MACRO” menu item. After that, a gray spot appears near this vertex and the caption is displayed with a bold font

For example:



To call a macro in the program, write its name into the “FIELD” element, similar to a subroutine call.

Furthermore, the compiler allows the creation the user macros with parameters. For that, the parameters must be placed, separated by commas, after the caption in the round brackets. In the operators of the macro body, the “ ~ “ character has only the separating function, and the compiler ignores it.

When calling such macro, the compiler replaces the declared in the macro caption names to names of a macro call. For example:

```

|
...      A1~L Op A2~L
...      A1~H Op A2~H Op
WordOperation(X,+,Y)
...
...

```

In this example instead of:

```
WordOperation(X,+,Y)
```

macro call, the compiler places:

```
XL + YL
XH + YH +
```

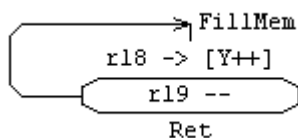
operators.

When the compiler finds an error in a macro, it shows the erroneous operator. To show a call of this macro, select the “**Search/Show macro call**” menu item.

Subroutines with parameters

Before being called, subroutines often require preliminary storing of initial values. As a rule, the working registers or SRAM variables are used for this purpose. In this case, the operators of loading these initial values should be placed prior to the call. But such method is inconvenient and not visual. Using the macros, Algorithm Builder allows the forming the call of such subroutines in the handy form, placing initial parameters into the brackets, as common with high level languages.

In the below represented example, the "FillMem" subroutine fills the a fragment of SRAM by appointed value:



This subroutine requires storing the preliminary values:

in the Y – initial address of the SRAM fragment;

in the r19 – number of filled bytes;

in the r18 – filled value.

The subroutine call in the usual way is:

```

$200 -> Y
16 -> r19
$22 -> r18
FillMem

```

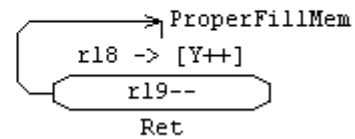
This call provides storing of \$22 to 16 bytes, starting from the address \$200.

For the possibility of a subroutine call with parameters, it should be created together with a macro. For example:

```

    ● FillMem(Value, Address, Count)
Value -> r18
Address -> Y
Count -> r19
ProperFillMem

```



In this case, the call of such a subroutine will be:

```

    ...
    FillMem($22, $200, 16)
    ...

```

Instead of constants, the parameters can be a working registers, SRAM or EEPROM variables, I/O registers etc.

The floating-point data format

The Algorithm Builder supports the working with floating-point values. These values can be represented either in 32-bits format or in 64-bits (double precision).

The 32-bit format:

1	8	23
S	E	F

The value V of the number is given by:

if $0 < E < 255$, then $V = (-1)^S \cdot 2^{(E-127)} \cdot (1.F)$

if $E=0$ and $F=0$, then $V = (-1)^S \cdot 0$

if $E=255$ and $F=0$, then $V = (-1)^S \cdot \text{INF}$ (infinity)

if $E=255$ and $F \neq 0$, then V is a NAN (not a number)

The 64-bit format:

1	11	52
S	E	F

The value V of the number is given by

if $0 < E < 2047$, then $V = (-1)^S \cdot 2^{(E-1023)} \cdot (1.F)$

if $E=0$ and $F=0$, then $V = (-1)^S \cdot 0$

if $E=2047$ and $F=0$, then $V = (-1)^S \cdot \text{INF}$ (infinity)

if $E=2047$ and $F \neq 0$, then V is a NAN (not a number)

Floating-point constant must have a decimal point and (or) exponent of 10 after the "E" or "e" character. For example: "1.27", "255.99E-22", "5e12".

By default, these constants are in 32-bit format. To define a 64-bits constant, add the typecast ":Int64" or ":QWord". For example: the "349.85" constant is interpreted as a 32-bit value \$43AECCD and the "349.85:QWord" constant is interpreted as a 64-bit value \$4075DD999999999A.

Note the copying is only one correctly standard macro-operation for suchlike values. For example, when the "A32" and "B32" SRAM variables are declared, they are admissible the following operations:

```

348.85 -> A32          (equal to: "$43AECCD -> A32")
A32 -> B32

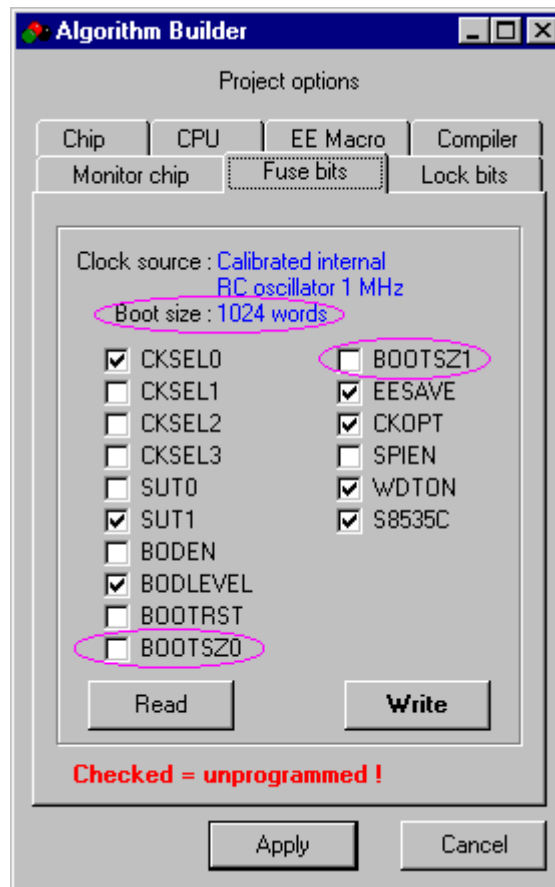
```

To perform mathematical operations, use the appropriate subroutines. Some of such subroutines are in the "Float32.alg" file in the "EXAMPLE" directory.

Programming the Boot Loader

To program the boot loader, use the "BOOT:" directive in a "TEXT" element. So, the compiler will place all following operators into the boot section.

The "BOOTSZ" fuse bits in the project options define the initial address and the size of this section:



To handle interrupts in the boot loading section, use interrupt names with the "BOOT_" prefix.

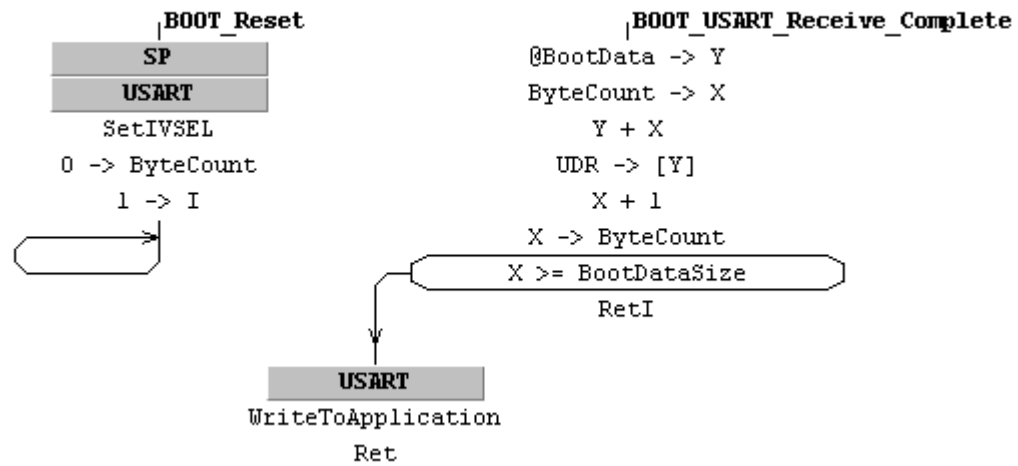
When the BOOTRST fuse bit is programmed (cleared), the program execution will start in the boot section. To switch over the interrupts to this section, set the IVSEL bit. To change this bit, use the in-built macro-operation SetIVSEL and ClearIVSEL.

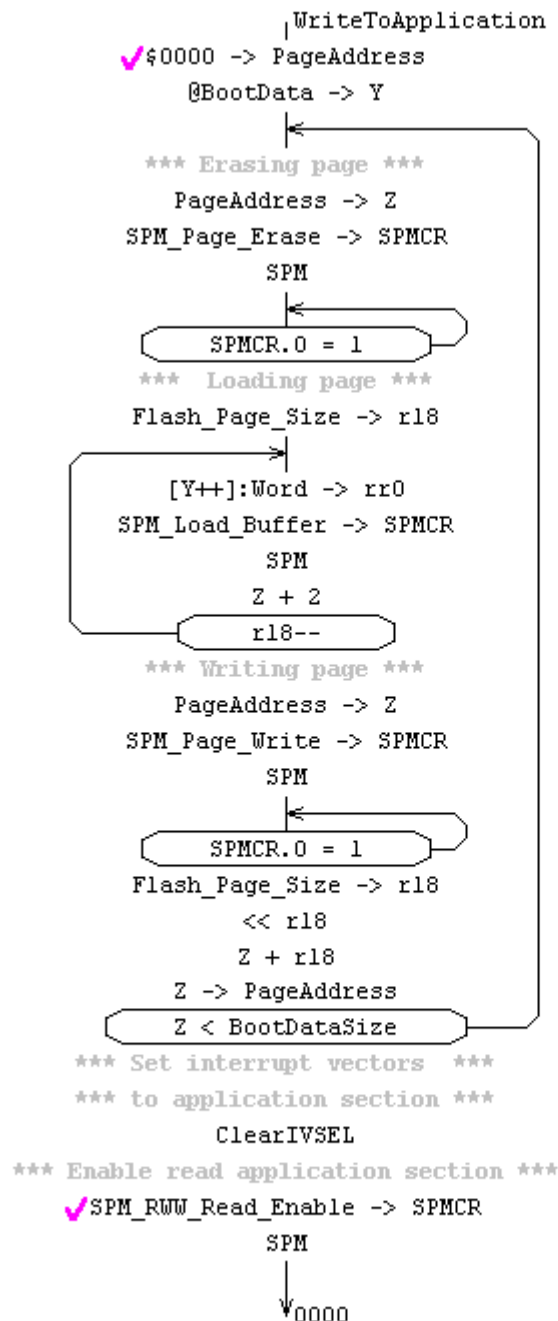
By default, the following constants are declared in the Algorithm Builder:

```
SPM_Load_Buffer      = #b00000001
SPM_Page_Erase       = #b00000011
SPM_Page_Write       = #b00000101
SPM_Buffer_Erase     = #b00010001
SPM_RWW_Read_Enable = #b00010001
```

Example:

BOOT:





The completely this algorithm is in the "EXAMPLES\BOOT RECEIVER" directory.

Note the "BOOT:" directive is active at a current page only. When it is necessary to switch over the compilation to the application section, use the "Application:" directive. By default, the page is compiled into the application section.

Information about modified Working registers

Developing an algorithm, it is important to know about modified working registers by a subroutine call or by a macro. A bubble help on a name of a subroutine or a macro contains this information. At that, it contains an enumeration of the modified and NOT modified working registers. But, for the appearing of this list, the project must be compiled successfully.

If there is an indirect subroutine call in the body of a subroutine or macro, the list of modified working registers will be not exact. In this case the hint will contain the respective comment.

Project files

The leftmost page of the algorithm, created in the editor, is saved on the disk with the ".alp" extension, and the other pages – with the "*.alg" extension. When opening the files in the editor, a copy with ".~al" extension is created which may be used to restore a source file.

The configuration and environment of the project, with the names of the most recently opened files; the position and size of the windows and other information of the project are saved in an ".ini" file with the same as the ".alp" file. After a successful compilation the program memory and EEPROM files are created on the hard drive. The program memory file has the same name as the ".alp" file and the EEPROM file has the "EE_" prefix added to it.

The files can be created in a binary format with the ".bin" extension, in a textual "Generic" format with the ".gen" extension or in an "Intel HEX" format with the ".hex" extension. To select the format use "Options\Environment Options..." menu item.

Uninstalling the “Algorithm Builder”

To uninstall, choose the "Options\Uninstall..." menu item. Then follow the instructions in the displayed windows.

Note, only the current version of a product is uninstalled.