

Графическая среда разработки программного обеспечения для микроконтроллеров с архитектурой AVR “Algorithm Builder”

Данная среда обеспечивает полный цикл разработки, начиная от ввода алгоритма, включая отладку, и заканчивая внутрисхемным программированием кристалла. Вы будете иметь возможность разрабатывать программы как на уровне ассемблера, так и на макро-уровне, при котором возможна работа со знакопеременными величинами произвольной длины. Это приближает возможности программирования к языку высокого уровня.

Графические технологии создания программы раскрывают новые возможности для программистов. Они позволяют вводить программы на плоскости в виде алгоритма с древовидной структурой. В результате вся логическая структура программы становится полностью наглядной. Основным предназначением таких технологий является максимальное приведение интерфейса разработки к природе человеческого восприятия. Освоение такой среды намного проще, чем освоение классического ассемблера. Более удобный интерфейс раскрывает новые возможности для разработки. По оценке пользователей, время создания программного обеспечения сокращается в 3 - 5 раз по сравнению с классическим ассемблером.

Среда предназначена для работы под ОС Windows 95/98/2000/NT/ME/XP.
Для нормальной работы редактора требуется наличие шрифта “Courier”.

Конструкция алгоритма

Любое программное обеспечение можно разбить на отдельные логически завершенные фрагменты. Как правило, финальным оператором этих фрагментов являются такие операторы как безусловный переход или возврат из подпрограммы, т.е. операторы, после которых линейное исполнение программы однозначно прекращается.


Разработка программного обеспечения в среде Algorithm Builder сводится к формированию таких блоков, размещению их на плоскости и установлению между ними векторных связей из условных и безусловных переходов.

Элементы алгоритма

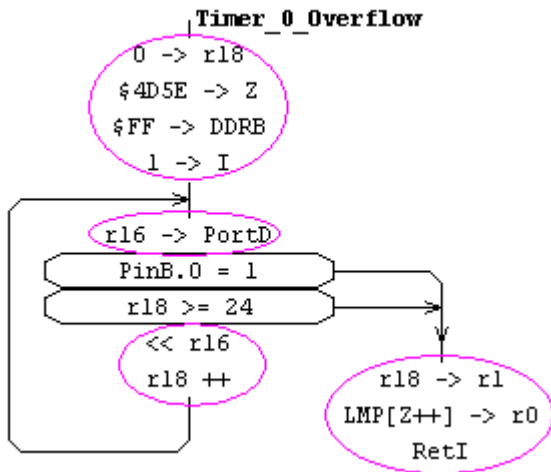
Для построения алгоритма в Algorithm Builder предусмотрено семь элементов:

FIELD	–	Поле;
LABEL	–	Метка;
VERTEX	–	Вершина блока;
CONDITION	–	Условный переход;
JMP Vector	–	Относительный безусловный переход;
SETTER	–	Настройщик периферийных устройств;
TEXT	–	Строка локального текстового редактора.


Элемент «FIELD» - поле

Представляет собой отцентрированную в блоке строку. Объект предназначен для записи большинства операторов микроконтроллера. Для того чтобы добавить поле, выберите либо пункт меню “Elements\Field”, либо нажмите кнопку  на панели инструментов, либо клавишу “Alt+F”, либо клавишу “Enter” (если курсор находится вне локального текстового редактора).

В представленном ниже примере алгоритма элементы "Field" обведены овалами.

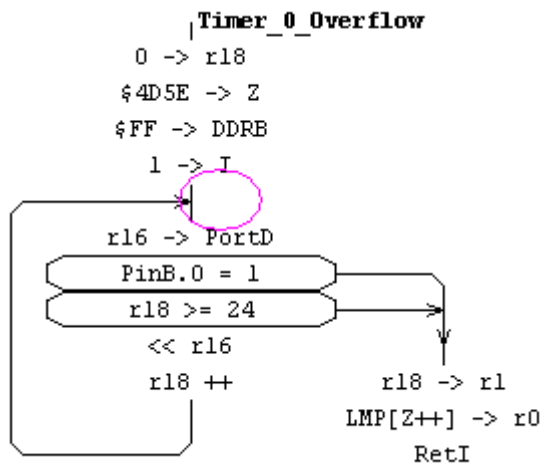


Элемент «LABEL» - метка

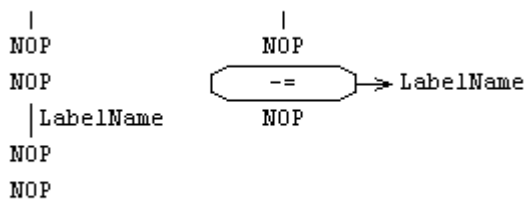
Представляет собой вертикальный штрих, расположенный внутри блока операторов и необязательное имя, располагающееся слева или справа от штриха. Предназначена метка для обозначения мест в алгоритме, куда возможно осуществление условных и безусловных переходов. Для добавления метки в блок выберите пункт меню "Elements\Label", нажмите клавиши "Alt+L" или кнопку  на панели инструментов. При необходимости можете назначить конкретный адрес программы. Для этого, перед именем (если оно есть) необходимо записать константу или алгебраическое выражение, которое определяет этот адрес.

Для изменения расположения имени метки на противоположное нажмите клавишу "Tab".

В приведенном ниже примере элемент "Label" обведен овалом.




Как правило, к метке должен примыкать один из векторов перехода (со стрелкой на конце). В этом случае имя метке давать необязательно. Но Algorithm Builder допускает классическую адресацию переходов с использованием имен меток. В этом случае метке имя дать необходимо. Например:

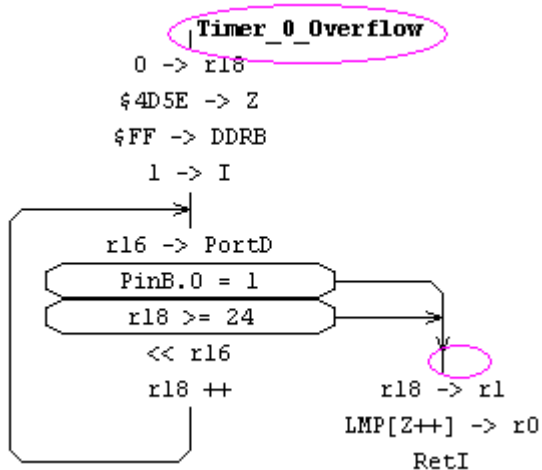


Кроме того, в операторах алгоритма имя метки может быть использовано как константа, содержащая адрес соответствующего места в памяти программы.

Элемент «VERTEX» - вершина блока

По своему отображению и назначению полностью идентичен метке, но, в отличие от нее, задает расположение блока на рабочей плоскости и всегда является его началом. Для того, чтобы добавить новую вершину либо выберите пункт меню “Elements\Vertex”, либо нажмите клавиши “Alt+V”, либо нажмите кнопку  на панели инструментов, либо левую кнопку мыши, нажав ее на необходимом месте рабочего поля в комбинации с клавишами “Alt+Ctrl+Shift”.

В приведенном ниже примере элементы “Vertex” обведены овалами:




Как правило, имя вершине назначается, только если она является началом подпрограммы или макроса.

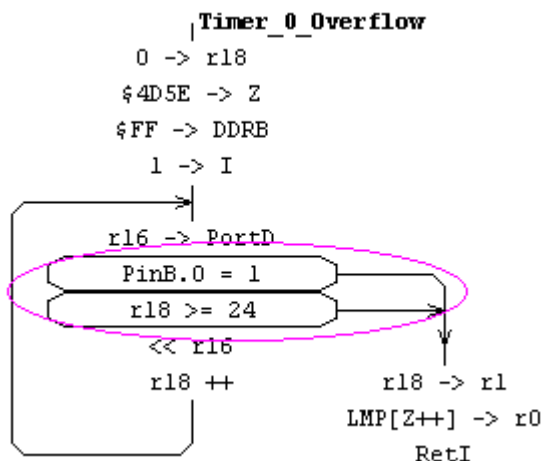
Элемент «CONDITION» - условный переход

Этот элемент конструктивно наиболее сложный, предназначен для реализации условных переходов. Представляет собой овальный контур, внутри которого вписывается условие перехода и возможный вектор в виде ломаной линии со стрелкой на конце, возле которой возможно необязательное имя вектора.

Конец вектора должен либо заканчиваться на какой-либо метке, или вершине, либо на отрезке другого вектора, либо иметь имя адресуемой метки.

Для того чтобы провести вектор до нужного места, нажмите левую кнопку мыши при нажатой клавише “Alt”. Для редактирования вектора используйте клавиши направления в комбинации с клавишей “Alt”. Для перехода от редактирования условия к редактированию имени вектора нажмите клавишу “Tab”. Последующие нажатия клавиши “Tab” будут менять положение имени вектора на противоположное. Чтобы ввести новый объект, либо выберите пункт меню “Elements\Condition”, либо нажмите клавиши “Alt+C”, либо кнопку  на панели инструментов.

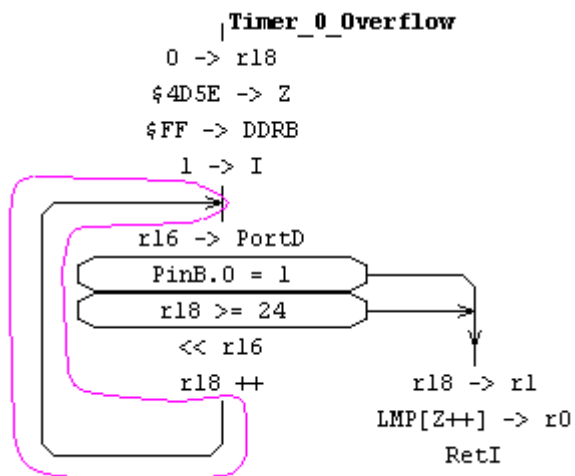
В приведенном ниже примере элементы “Condition” обведены овалами:



Элемент «JMP Vector» - безусловный переход

Этот элемент предназначен для реализации коротких безусловных переходов (в базовом ассемблере это оператор “RJMP”). Представляет собой ломаную линию, исходящую из середины блока со стрелкой на конце, аналогичную вектору объекта “Condition”. Чтобы добавить новый безусловный переход, либо выберите пункт меню “Elements\JMP Vector”, либо нажмите клавиши “Alt+J”, либо кнопку **J** на панели инструментов.

В приведенном ниже примере элемент “JMP Vector” обведен линией.



Элемент «SETTER» - настройщик

Этот объект представляет собой серый прямоугольник, внутрь которого вписано имя настраиваемого периферийного компонента микроконтроллера, такого как Таймер, АЦП, регистр маски прерываний и пр. Настройщик предназначен для формирования последовательности операций микроконтроллера, которые обеспечивают загрузку необходимых констант в соответствующие управляющие регистры ввода-вывода в соответствии с выбранными свойствами.

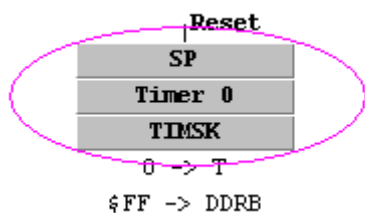
Перед использованием этого элемента тип микроконтроллера должен быть определен (пункт меню “Options\Project options”, закладка “Chip”)

Для добавления в алгоритм настройщика, либо выберите пункт меню “Elements\Setter”, либо нажмите клавиши “Alt+S”, либо кнопку **S** на панели инструментов. Для редактирования уже введенного настройщика, активизируйте редактор двойным щелчком мыши, либо клавишами “Shift+Enter”.

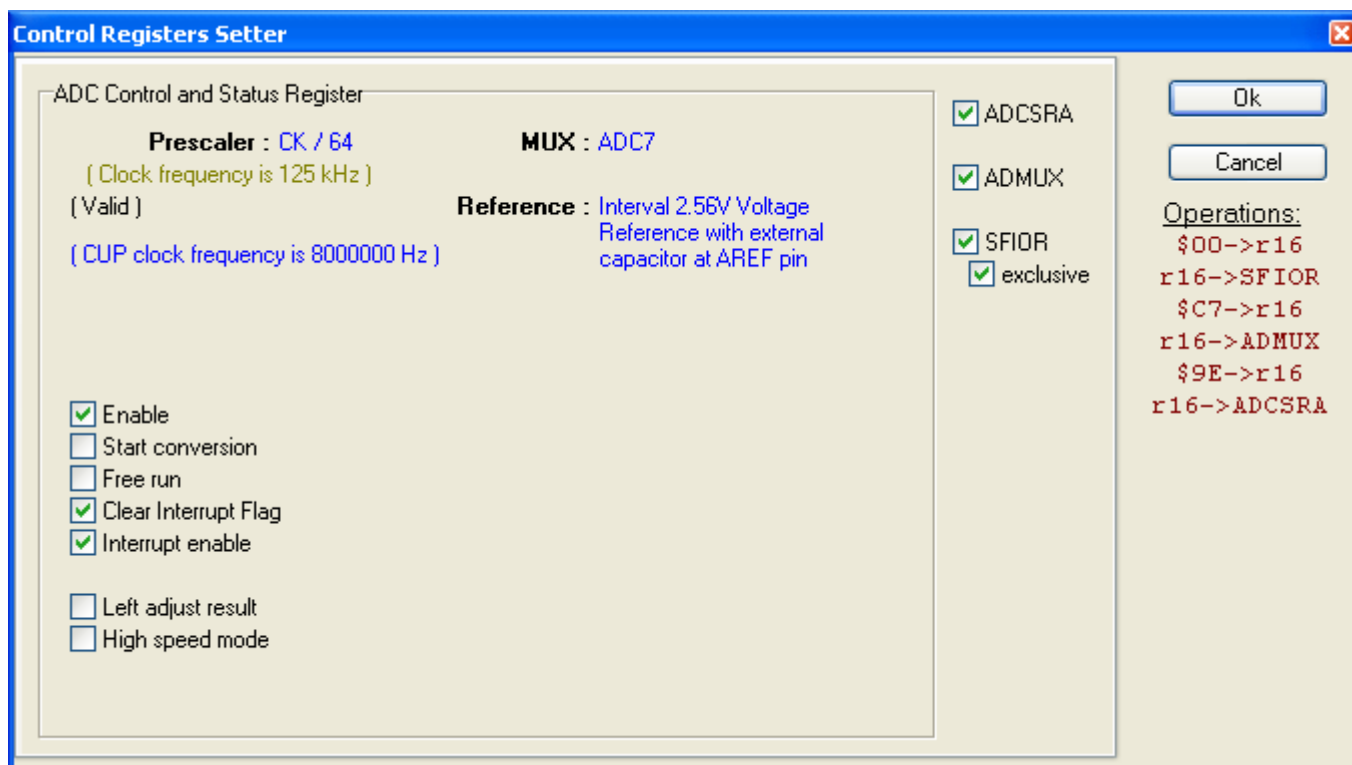
Объект “Setter” является макро-оператором. После компиляции он преобразуется в последовательность команд микроконтроллера, которые обеспечат загрузку необходимых констант в соответствующие управляющие регистры. При этом следует иметь в виду, что в этих операциях будет использован регистр-посредник r16.

Для ряда компонент, например ADC, настройщик может воздействовать на несколько управляющих регистров. В этом случае при необходимости воздействие на каждый конкретный регистр можно заблокировать.

В приведенном ниже примере элементы “Setter” обведены овальной линией.



Пример раскрытого окна настройщика АЦП для ATmega128:




В правой стороне, под кнопками, отображается набор генерируемых настройщиком операций. На правой стороне рабочей страницы располагается вертикальный ряд ключей активизации используемых управляющих регистров, позволяющие воздействовать регистры выборочно.

Некоторые ключи активизации регистров имеют опцию “exclusive”. Наличие этой опции означает, что данный регистр используется не только одним этим компонентом (в приведенном примере, регистр “SFIO” использует не только АЦП). При этом, если опция включена, то в незадействованные компонентой (в примере – АЦП) биты будут записаны нули. При отключенной опции, в регистре будут модифицированы только используемые биты, а остальные останутся без изменений. Это гарантирует, что настройщик не нарушит уже существующую настройку другого компонента, биты которого располагаются в том же регистре, но в этом случае генерируемый код будет длиннее.

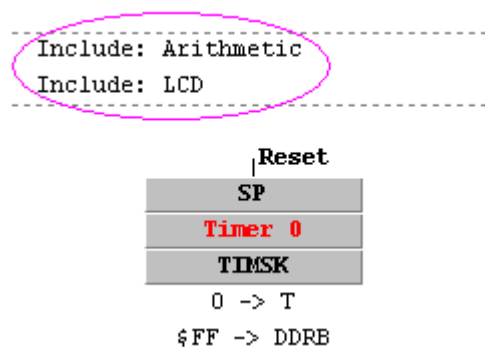
Элемент «ТЕХТ» - строка локального текстового редактора

Этот элемент представляет собой текстовую строку, начинающуюся от левого края поля алгоритма. Совокупность из нескольких таких строк образует локальный текстовый редактор, окаймленный пунктирными линиями. Правила работы в нем, аналогичны прочим текстовым редакторам.

Строки предназначены для записи в них ряда директив компилятора, а также для комментариев. Для добавления нового локального текстового редактора либо выберите пункт меню “Elements\Text”, либо нажмите клавиши “Alt+T”, либо кнопку  на панели инструментов.

Комментарии должны начинаться с двух косых: “//”.

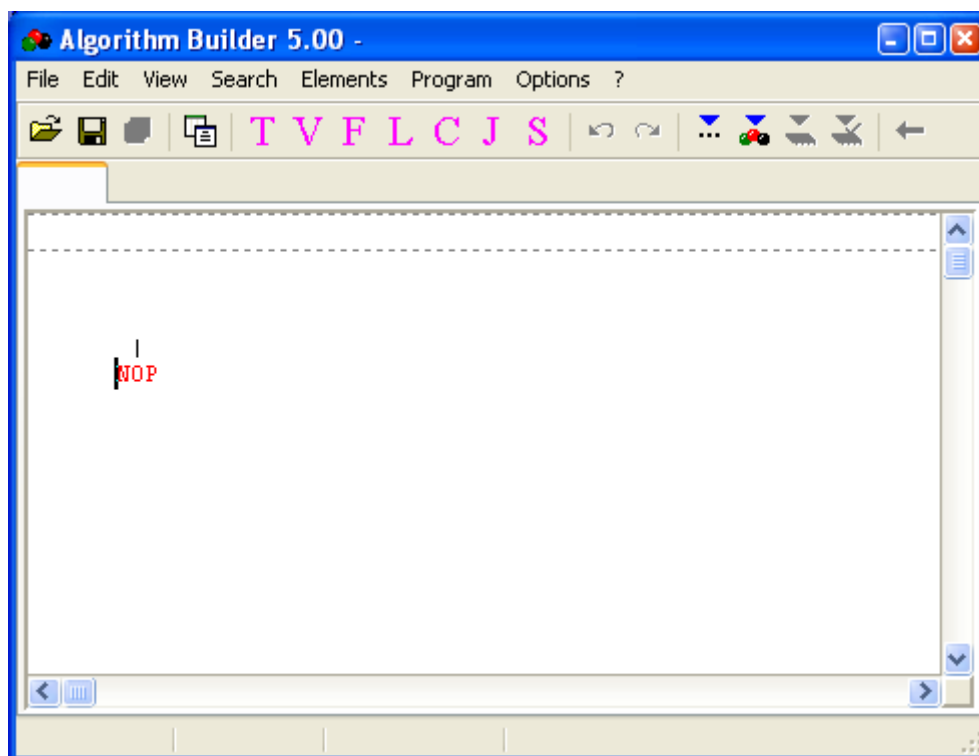
В приведенном ниже примере элементы "Text" обведены овальной линией.



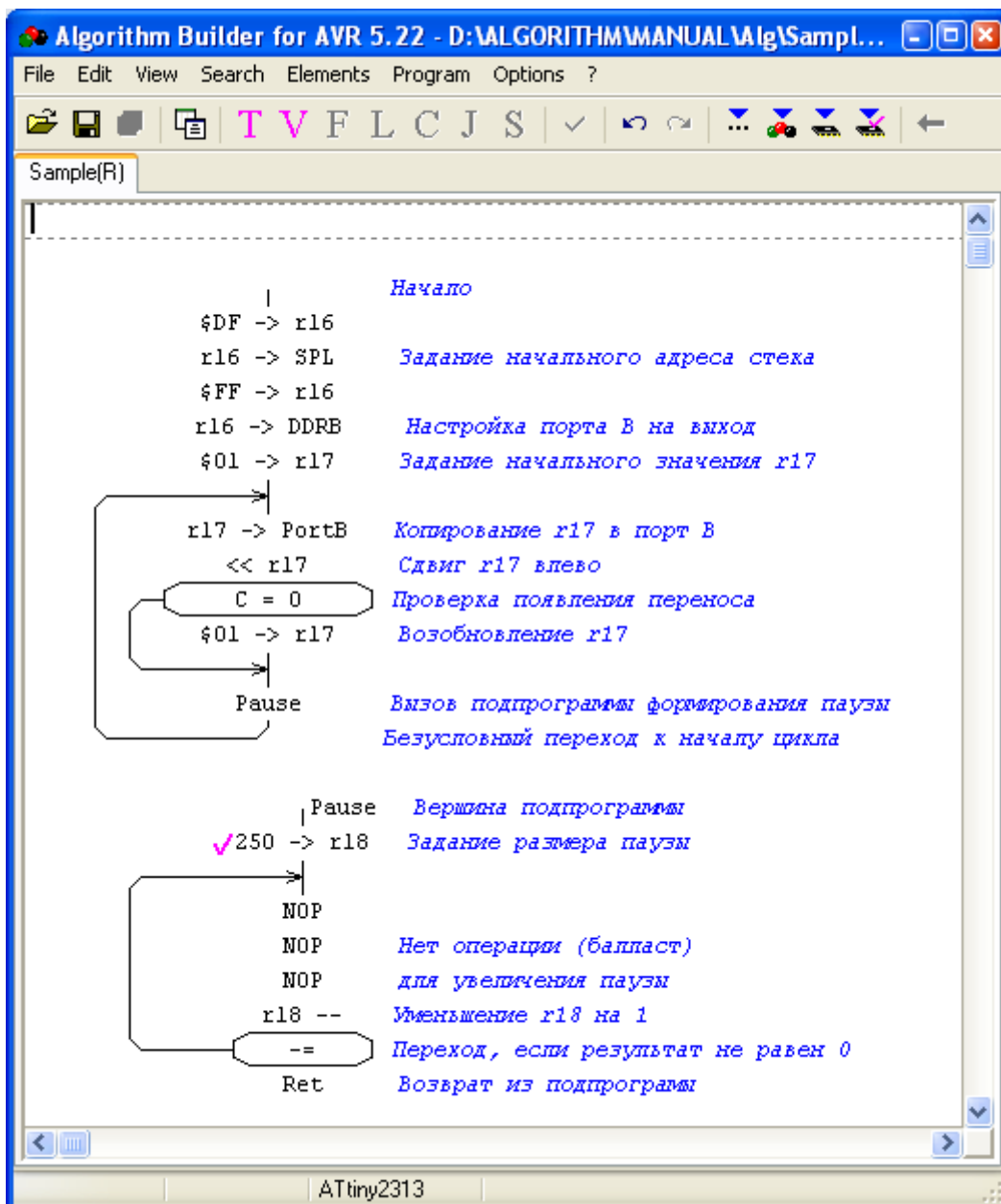
Пример простого алгоритма

Если в данный момент уже загружен какой-либо алгоритм, то его необходимо закрыть. Для этого выберите пункт меню "File\Close project".

Для создания нового проекта выберите пункт меню "File\New". При этом появится первая закладка и на рабочем поле элемент "TEXT", "VERTEX" и прикрепленный к нему один элемент "FIELD" с пустым оператором "NOP":



Введите приведенный ниже пример простого алгоритма:



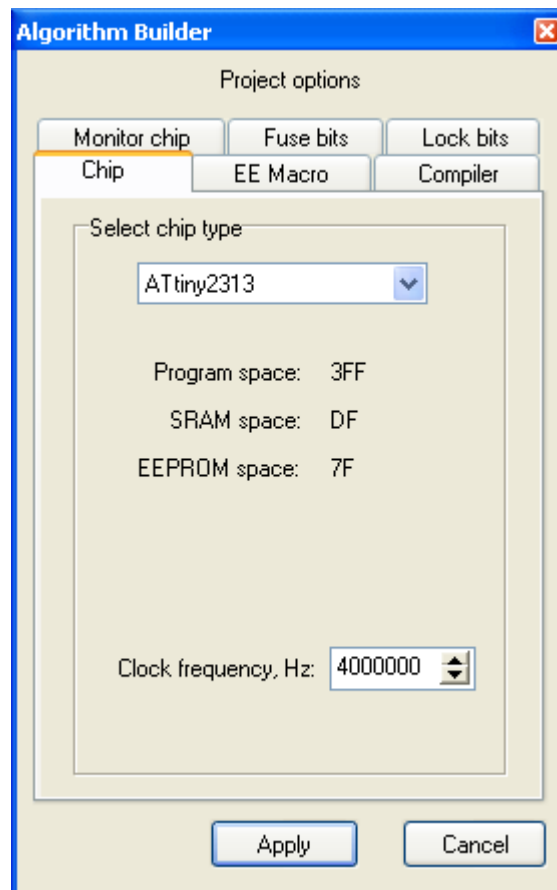
Этот алгоритм поочередно выводит логические единицы в биты порта В. Алгоритм содержит только элементарные инструкции микроконтроллера. Данный пример содержится в прилагаемой директории "Sample0(R)", но целесообразнее ввести его вручную.


В операторах копирования часто используются символы "->". Для удобства набора, их можно ввести нажатием одной клавиши "~" (под клавишей "Escape"). Для добавления очередного элемента "FIELD" можно просто нажимать клавишу "Enter", метки – клавиши "Alt+L", условного перехода – "Alt+C", безусловного перехода – "Alt+J". Альтернативно эти элементы можно добавлять соответствующими кнопками на панели инструментов.

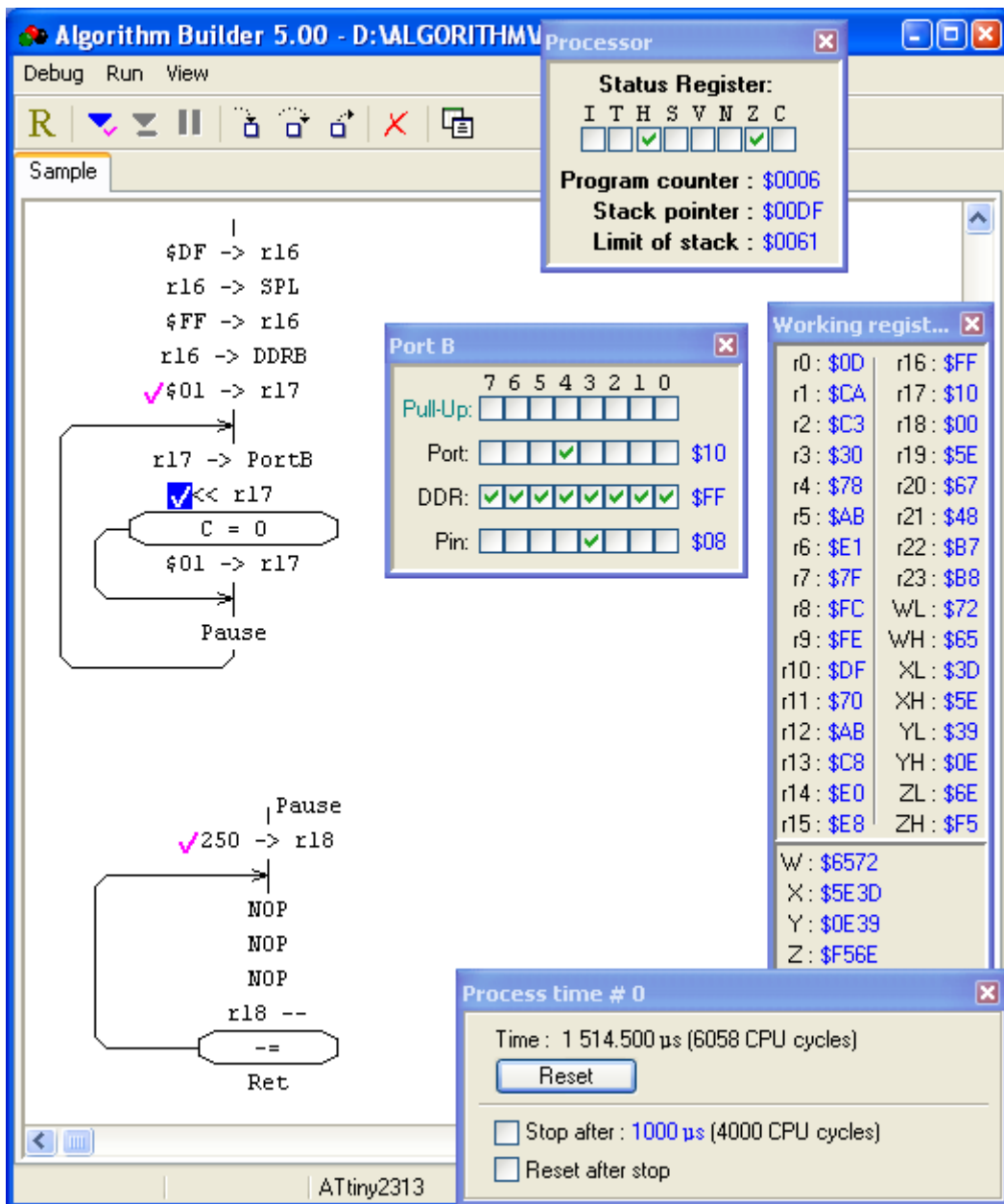
Для добавления в алгоритм нового блока, поместите фокус редактора на последний элемент предыдущего блока и нажмите клавиши "Alt+V".

Пурпурная "птичка" ✓ возле оператора "250->r18" означает точку останова для симулятора. Для ее добавления (или наоборот), поместите фокус редактора на этот оператор и нажмите клавишу "F5".

Перед запуском на компиляцию необходимо задать тип микроконтроллера. Для этого выберите пункт меню "Options\Project options...":



Для исполнения алгоритма в симуляторе нажмите клавишу "F9" или кнопку  на панели инструментов. При этом в начале произойдет компиляция и, если отсутствуют ошибки, запустится симулятор. Открытие необходимых окон компонент микроконтроллера делается через пункт меню "View...". Для наблюдения процессов во введенном алгоритме достаточно открыть окна "Processor", "PortB" и "Working registers".



Метка возле оператора означает текущее положение программного счетчика. Метка располагается перед оператором, который должен быть исполнен в следующем шаге. Сразу после запуска, она располагается на первом операторе программы, располагающемся по адресу \$0000.

При запуске симулятора содержимое рабочих регистров и SRAM всегда заполняется случайными величинами, поскольку в реальном микроконтроллере их содержимое после подачи питания непредсказуемо.

Пошаговое исполнение алгоритма с входом в подпрограммы осуществляется кнопкой "F7" или кнопкой на панели инструментов. Пошаговое исполнение без входа в подпрограммы – клавишей "F8" или кнопкой . Исполнение алгоритма до выхода из подпрограммы – клавишей "F6" или кнопкой .

Запуск на непрерывное исполнение до точки останова () осуществляется клавишей "F9" или кнопкой , а до выделенного элемента – клавишей "F4" или кнопкой . При этом исполнение алгоритма может быть приостановлено нажатием клавиши "F2" или кнопки .

В процессе исполнения данного алгоритма можно наблюдать поочередное появление логических единиц на выходах порта B.

При желании Вы можете запрограммировать этот алгоритм в реальный кристалл и наблюдать осциллографом работу.

Операторы микроконтроллера

Система команд AVR содержит более 130 элементарных операций. В приведенных ниже таблицах приведены все имеющиеся операции.

Операции копирования и арифметико-логических преобразований

Для их записи используются только элементы "FIELD".

Шаблон	Действие	Пример	Аналог
R -> R	Копирование одного рабочего регистра в другой	R0 -> R1	MOV R,R
# -> R	Загрузка константы # в рабочий регистр	24 -> r16	LDI R,K
[X] -> R	Копирование из SRAM в рабочий регистр косвенно по X	[X] -> R2	LD R,X
[X++] -> R	Копирование из SRAM в рабочий регистр косвенно по X с постинкрементом	[X++] -> R3	LD R,X+
[--X] -> R	Копирование из SRAM в рабочий регистр косвенно по X с предекрементом	[--X] -> R4	LD R,-X
[Y] -> R	Копирование из SRAM в рабочий регистр косвенно по Y	[Y] -> R5	LD R,Y
[Y++] -> R	Копирование из SRAM в рабочий регистр косвенно по Y с постинкрементом	[Y++] -> R6	LD R,Y+
[--Y] -> R	Копирование из SRAM в рабочий регистр косвенно по Y с предекрементом	[--Y] -> R7	LD R,-Y
[Y+#] -> R	Копирование из SRAM в рабочий регистр косвенно по Y со смещением #	[Y+2] -> r8	LDD R,Y+q
[Z] -> R	Копирование из SRAM в рабочий регистр косвенно по Z	[Z] -> R9	LD R,Z
[Z++] -> R	Копирование из SRAM в рабочий регистр косвенно по Z с постинкрементом	[Z++] -> R10	LD R,Z+
[--Z] -> R	Копирование из SRAM в рабочий регистр косвенно по Z с предекрементом	[--Z] -> R11	LD R,-Z
[Z+#] -> R	Копирование из SRAM в рабочий регистр косвенно по Z со смещением #	[Z+1] -> R12	LDD R,Z+q
[#] -> R	Копирование из SRAM непосредственно из адреса # в рабочий регистр	[\$60] -> r14	LDS R,k
R -> [X]	Копирование рабочего регистра в SRAM косвенно по X	R15 -> [X]	ST X,R
R -> [X++]	Копирование рабочего регистра в SRAM косвенно по X с постинкрементом	R16 -> [X++]	ST X+,R
R -> [--X]	Копирование рабочего регистра в SRAM косвенно по X с предекрементом	R17 -> [--X]	ST -X,R
R -> [Y]	Копирование рабочего регистра в SRAM косвенно по Y	R18 -> [Y]	ST Y,R
R -> [Y++]	Копирование рабочего регистра в SRAM косвенно по Y с постинкрементом	R19 -> [Y++]	ST Y+,R
R -> [--Y]	Копирование рабочего регистра в SRAM косвенно по Y с предекрементом	R20 -> [--Y]	ST -Y,R

	SRAM косвенно по Y с пре-декрементом		
R -> [Y+#]	Копирование рабочего регистра в SRAM косвенно по Y со смещением #	R21 -> [Y+2]	STD Y+q,R
R -> [Z]	Копирование рабочего регистра в SRAM косвенно по Z	R22 -> [Z]	ST Z,R
R -> [Z++]	Копирование рабочего регистра в SRAM косвенно по Z с пост-инкрементом	R23 -> [Z++]	SR Z+,R
R -> [--Z]	Копирование рабочего регистра в SRAM косвенно по Z с пре-декрементом	R24 -> [--Z]	ST -Z,R
R -> [Z+#]	Копирование рабочего регистра в SRAM косвенно по Z со смещением #	R25 -> [Z+3]	STD Z+q,R
R -> [#]	Копирование регистра в SRAM непосредственно по адресу #	R26 -> [\$200]	STS k,R
LPM LPM[Z]	Загрузка из памяти программы по адресу Z в r0		LPM
LPM -> R LPM[Z] -> R	Загрузка из памяти программы по адресу Z в рабочий регистр		LPM R,Z
LPM[Z++] -> R	Загрузка из памяти программы по адресу Z в рабочий регистр и пост-инкремент Z	LPM[Z++] -> r5	LPM R,Z+
SPM	Запись в память программы		SPM
P -> R	Копирование I/O регистра в рабочий регистр	P\$19 -> WH	IN R,P
R -> P	Копирование рабочего регистра в I/O регистр	XL -> PortB	OUT P,R
R->	Копирование рабочего регистра в стек	r0->	PUSH R
->R	Копирование из стека в рабочий регистр	->r1	POP R
NOP	Нет операции		NOP
R + R	Прибавление к левому рабочему регистру правого	r0 + r1	ADD R,R
R + R +	Прибавление к левому рабочему регистру правого с переносом	R0 + R5 +	ADC R,R
RR + #	Увеличение двойного рабочего регистра (W, X, Y, Z) на #	W + 32	ADIW RW1,K
R - R	Вычитание из левого рабочего регистра правого	r4 - r5	SUB R,Rr
R - #	Вычитание из рабочего регистра константы #	XL - 2	SUBI R,K
R + #	Прибавление к рабочему регистру константы #	R17 + 12	SUBI R,256-K
R - R -	Вычитание из левого рабочего регистра правого с переносом	r5 - r6 -	SBC R,R
R - # -	Вычитание из рабочего регистра константы # с переносом	ZL - \$70 -	SBCI R,K
RR - #	Вычитание из двойного рабочего регистра (W, X, Y, Z) константы #	X - \$2E	SBIW RW1,K
R & R	Побитная логическая операция И двух рабочих регистров.	r7 & r8	AND R,R
R & #	Побитная логическая операция И рабочего регистра и константы #	r17 & #b00111101	ANDI R,K
R & #	Побитная логическая операция И рабочего регистра и инверсии константы #	r18 & #b00010011	CBR R,K
R ! R	Побитная логическая операция	R9 ! r10	OR R,R

	ИЛИ двух рабочих регистров.		
R ! #	Побитная логическая операция ИЛИ рабочего регистра и константы #	R18 ! #0147	ORI R,K
R ^ R	Побитная логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ двух рабочих регистров.	R11 ^ r12	EOR R,R
R	Побитная инверсия рабочего регистра	-r8-	COM R
-R	Арифметическая инверсия рабочего регистра	-RB	NEG R
R++	Увеличение рабочего регистра на 1	Count++	INC R
R--	Уменьшение рабочего регистра на 1	Idx--	DEC R
R?	Тест рабочего регистра (R & R)	R16?	TST R
^R	Очистка рабочего регистра (R ^ R)	R8	CLR R
R * R	Умножение двух рабочих регистров	R6 * R17	MUL R,R
±R * ±R	Умножение двух рабочих регистров с учетом знака	±r16 * ±r20	MULS R,R
±R * R R * ±R	Умножение двух рабочих регистров учетом знака одного из регистров.	±r18 * r22 r22 * ±r18	MULSU R,R
<<(R * R)	Дробное умножение двух рабочих регистров	<<(R6 * R17)	FMUL R,R
<<(±R * ±R)	Дробное умножение двух рабочих регистров с учетом знака	<<(±r16 * ±r20)	FMULS R,R
<<(±R * R) <<(R * ±R)	Дробное умножение двух рабочих регистров с учетом знака одного из регистров	<<(±r18 * r22) <<(r22 * ±r18)	FMULSU R,R
1 -> P.#	Запись 1 в бит # I/O регистра	1 -> DDB2	SBI P,b
0 -> P.#	Запись 0 в бит # I/O регистра	0 -> PortD.7	CBI P,b
<<R	Логический сдвиг влево рабочего регистра	<<r18	LSL R
R>>	Логический сдвиг вправо рабочего регистра	R19>>	LSR R
<<R<	Логический сдвиг влево рабочего регистра с переносом	<<r20<	ROL R
>R>>	Логический сдвиг вправо рабочего регистра с переносом	>r21>>	ROR R
±R>>	Арифметический сдвиг вправо рабочего регистра	±r22>>	ASR R
>>R<<	Обмен тетрад рабочего регистра	>>r23<<	SWAP R
R.#->	Копирование бита # рабочего регистра в T (SREG)	RA.6->	BST R,b
->R.#	Копирование T (SREG) в бит # рабочего регистра	->r0.4	BLD R,b
# -> R.#	Запись # (0 or 1) в # бит рабочего регистра	0 -> r18.4 1 -> r20.0	ANDI R,256-K ORI R,K
RR -> RR	Копирование двойного рабочего регистра в двойной рабочий регистр	Y -> X	MOVW R,R
1 -> C	Запись 1 в бит C		SEC
0 -> C	Запись 0 в бит C		CLC
1 -> N	Запись 1 в бит N		SEN
0 -> N	Запись 0 в бит N		CLN
1 -> Z 1 -> .Z	Запись 1 в бит Z		SEZ
0 -> Z 0 -> .Z	Запись 0 в бит Z		CLZ

1 -> I	Запись 1 в бит I		SEI
0 -> I	Запись 0 в бит I		CLI
1 -> S	Запись 1 в бит S		SES
0 -> S	Запись 0 в бит S		CLS
1 -> V	Запись 1 в бит V		SEV
0 -> V	Запись 0 в бит V		CLV
1 -> T ; 1->	Запись 1 в бит T		SET
0 -> T ; 0->	Запись 0 в бит T		CLT
1 -> H	Запись 1 в бит H		SEH
0 -> H	Запись 0 в бит H		CLH
SLEEP	Приостановка работы кристалла		SLEEP
WDR	Сброс сторожевого таймера		WDR
JMP #	Дальний безусловный переход	JMP LabelName	JMP k
#/	Дальний вызов подпрограммы	LabelName/	CALL k
JMP[Z]	Косвенный безусловный переход по Z		IJMP
#	Относительный вызов подпрограммы	LabelName	RCALL k
CALL[Z]	Косвенный вызов подпрограммы по Z	Call[Z]	ICALL
RET	Возврат из подпрограммы	RET	RET
RETI	Возврат из подпрограммы и запись 1 в бит I	RETI	RETI
R = R	Сравнение двух рабочих регистров	R2 = R5	CP R,R
R = R =	Сравнение двух рабочих регистров с переносом	R6 = R5 =	CPC R,R
R = #	Сравнение рабочего регистра с константой #	YL = 47	CPI R,K

Во всех операциях умножения результат помещается в двойной регистр R0,R1.

Следует также обратить внимание на то, что любые операции между рабочим регистром и константой (например "#->R" или "R & #") возможны только для r16..r31, а операции с битами I/O регистров (например "1->PortB.4") – только для p0..p31.

Операторы условных переходов

Для реализации этих операторов используются элементы "CONDITION", в овал которого могут быть вписаны следующие условия переходов.

Шаблон условия перехода	Комментарии	Пример	Аналог
=	Результат равен нулю (Z=1)		BREQ k
!=	Результат не равен нулю (Z=0)		BRNE k
C = 1			BRCS k
C = 0			BRCC k
>=	Больше или равно		BRSH k
<	Меньше		BRLO k
-	Отрицательный результат		BRMI k
+	Положительный результат		BRPL k
±>=	Больше или равно с учетом знака		BRGE k
±<	Меньше с учетом знака		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
-------	--	--------

Вектор используемого элемента “CONDITION” должен либо заканчиваться на метке или вершине, либо иметь имя адресуемой метки.

Операторы условного пропуска следующего оператора

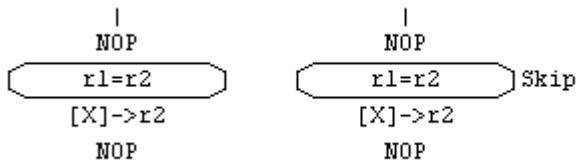
В этом случае вектор используемого элемента “CONDITION” либо отсутствует, либо имеет зарезервированное слово “Skip”.

Шаблон условия	Комментарии	Пример	Аналог
R = R	Равенство двух рабочих регистров	r0 = r1	CPSE R, R
R.# = 0	Бит # рабочего регистра равен 0	XH.2 = 0	SBRC R, b
R.# = 1	Бит # рабочего регистра равен 1	r14.3 = 1	SBRS R, b
P.# = 0	Бит # I/O регистра равен 0	PinC.6 = 0	SBIC P, b
P.# = 1	Бит # I/O регистра равен 1	EERE = 1	SBIS P, b

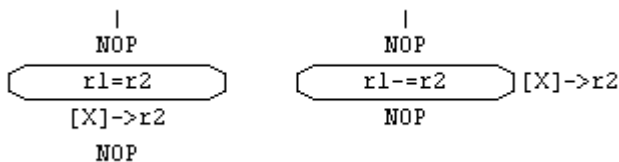
Здесь возможны два варианта конструкции элемента “CONDITION”:

1. Вектор элемента отсутствует, а имя вектора – либо отсутствует, либо содержит зарезервированное слово “Skip”. В нижеследующих примерах выполнение операции:

“[X]->r2” будет пропущено, если выполнится условие “r1=r2”.



2. Записывается инверсное условие, вектор отсутствует, а подлежащая пропуску инструкция вписывается как имя вектора. В этом случае объект будет интерпретирован как: выполнить инструкцию, если условие выполняется. Нижеследующие примеры полностью идентичны.



Комментарии к операторам

Окно с таблицей всех существующих для выбранного типа микроконтроллера шаблонов можно открыть через пункт меню “View\Templates...” и при необходимости, выбрав любой из них с помощью мыши, загрузить в редактируемый объект, нажав клавишу “Insert”.

Следует иметь в виду, что в некоторых типах микроконтроллеров может быть не весь набор операторов.

Символ “#” в шаблонах операторов означает константу. Они могут быть как положительными, так и отрицательными. Например, операторы:

-2 -> r16 и \$FE -> r16

будут откомпилированы с одинаковым результатом.

Следует обратить внимание на некоторые особенности реализации безусловных переходов и вызовов подпрограмм:

- Безусловный переход реализуется исключительно посредством элемента “JMP Vector”. При этом, если длина перехода не превышает 2047 пунктов, то компилятор применит короткий относительный переход (“RJMP”), в противном случае – длинный (“JMP k”).

- Вызов подпрограммы реализуется простой записью имени метки (вершины) или константы с адресом подпрограммы в элементе “FIELD”. При этом, если длина перехода не превышает 2047 пунктов, то компилятор применит короткий относительный вызов подпрограммы (“RCALL”), в противном случае – длинный (“CALL k”).

Например:

```

|
...
TestPins
...
|TestPins
...
Ret

```

- Косвенный переход (аналог - "IJMP") и вызов подпрограммы (аналог - "ICALL") реализуются соответственно записями "JMP[Z]" и "CALL[Z]" в элемент "FIELD" без параметров. Например:

```

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

```

Представление констант в непосредственном виде

В шаблонах операторов символ "#" означает константу, которая может быть представлена непосредственно.

Algorithm Builder поддерживает непосредственное представление констант в обычном, десятичном, шестнадцатеричном, восьмиричном, двоичном, символьном видах, а также в формате с [плавающей точкой](#).

В **десятичном** виде константа записывается в обычном виде с использованием символов "0..9", например: "35" или "24000".

В **шестнадцатеричном** виде константа начинается с символов: "\$", "#h" или "#H", затем используются символы "0..9" и "A..F" или "a..f". Например: "#h0F", "#He5", "\$23E7".

В **восьмиричном** виде константа начинается с символов: "#o" или "#O", затем используются символы "0..7". Например: "#o107", "#O2071".

В **двоичном** виде константа начинается с символов: "#b" или "#B", затем используются символы "0" и "1". Например: "#b01101101", "#B10011110".


В представлении константы допускается указание знака в виде символов "+" или "-". Например: "-54", "+2", "-\$5E3F".

При **символьном** представлении констант используются кавычки. В этом случае значением констант будут ANSI коды включенных в них символов. Например, такие представления констант, как "0" и \$30 будут полностью эквивалентны.

Если представление включает несколько символов, то такая константа будет многобайтной. Например, "12" будет эквивалентно константе \$3231.

Распределение ресурсов микроконтроллера и объявление имен

В Algorithm Builder программирование возможно с использованием стандартных имен рабочих регистров и регистров ввода-вывода, а обращение к ячейкам памяти SRAM и EEPROM производится непосредственно по физическому адресу. Но такое программирование неудобно.

Распределение ресурсов микроконтроллера и объявление имен производится в специализированной электронной таблице. Для переключения редактора между алгоритмом и таблицей выберите пункт меню "ViewToggle Algorithm/Data table" или нажмите клавишу "F12" или кнопку  на панели инструментов.

В объявляемых именах можно использовать буквы: "A..Z", "a..z", цифры "0..9", а также символ подчеркивания "_". Причем первым символом должна быть не цифра. Компилятор нечувствителен к регистру буквы, поэтому такие имена, как, например: "GenerateNoise", "GENERATENoise" и "generatenoise" будут считаться абсолютно одинаковыми.

Для распределения ресурсов и объявления имен, в электронной таблице предусмотрены шесть секций:

секция объявления имен рабочих регистров,
секция объявления имен регистров ввода-вывода,
секция объявления имен битов регистров,
секция объявления имен переменных SRAM,
секция объявления имен переменных EEPROM,
секция объявления имен констант.

Необходимость заполнения той или иной секции определяет сам программист.

Форматы переменных

Назначение форматов производится при объявлении имени группы рабочих регистров, регистров ввода-вывода, переменных SRAM, EEPROM, а также элементов массива констант, размещаемых в памяти программы.

В Algorithm Builder предусмотрены следующие форматы:

Формат	Ключевое слово
1 байт	"Int8" или "Byte"
2 байта	"Int16" или "Word"
3 байта	"Int24"
4 байта	"Int32" или "DWord"
5 байт	"Int40"
6 байт	"Int48"
7 байт	"Int56"
8 байт	"Int64" или "QWord"

Примечание: по умолчанию всегда принимается однобайтный формат.

Секция объявления констант

При программировании можно пользоваться константами в непосредственном виде. Но когда одна и та же константа используется в нескольких местах алгоритма, то целесообразнее дать ей имя и использовать его в операторах. При этом если потребуются изменить эту константу во всем алгоритме, то будет достаточно изменить только в месте ее объявления. Кроме того, использование имен существенно облегчает восприятие алгоритма.

Заголовок секции: **Constants:**

В этой секции предусмотрены следующие поля:

Name – объявляемое имя.

Value – алгебраическое выражение, определяющее величину;

Пример заполнения секции:

Constants:		
Name	Value	Commentary
Weight_Index	7	объявление Weight_Index константой 7
Weight_Min	10	
Weight_Typ	75	
Weight_Max	Weight_Min+100	
InputCode	§DC5E	

Для выборки байта из константы, к ее имени следует дописать ".#", где # – номер байта, начиная с 0. Например, оператор:

```
"InputCode.1 -> r16"
```

Загрузит в r16 константу §DC.

По умолчанию, в среде объявлены следующие системные константы:

"CPU_Clock_Frequency" – частота работы ядра в Герцах;

"IO_Org" – начальный IO регистров в адресном пространстве SRAM.

- “SRAM_Size” – размер SRAM в байтах;
- “SRAM_Org” – начало области SRAM;
- “EEPROM_Size” – размер EEPROM в байтах.
- “Flash_Size” – размер Flash памяти в 16-разрядных словах.
- “Flash_Page_Size” – размер страницы программирования Flash памяти в 16-разрядных словах.

Секция объявления имен рабочих регистров

Заголовок секции: **Working registers:**

В этой секции предусмотрены следующие поля:

- Name – объявляемое имя.
- Index – (необязательный параметр) константа, определяющая индекс регистра. По умолчанию принимается индекс, следующий за предыдущим, а в начале компиляции - равным нулю.
- Format – (необязательный параметр) **формат** объявляемого регистра. По умолчанию принимается однобайтный формат. При объявлении многобайтного регистра автоматически назначаются имена составляющих их однобайтных регистров. При объявлении двухбайтного регистра к именам добавляются буквы “L” и “H”, а для трех- и четырехбайтных – символы “0”, “1” и т.д. в соответствии с порядковым номером. Например, при объявлении двухбайтного (формат “Word”) регистра с именем “Counter”, автоматически объявляются составляющие его два однобайтных регистра с именами “CounterL” и “CounterH”. Многобайтные форматы рабочих регистров используются только в макро-операторах.

При объявлении допускается одному и тому же регистру давать несколько имен.

По умолчанию в среде действуют стандартные имена однобайтных регистров: “r0..r31” или “R0..R31”, а также: “WL”=“r24”, “WH”=“r25”, “XL”=“r26”, “XH”=“r27”, “YL”=“r28”, “YH”=“r29”, “ZL”=“r30”, “ZH”=“r31” и двухбайтных (“Int16” или “Word”): “W”= (“r24, r25”), “X”= (“r26, r27”), “Y”= (“r28, r29”) и “Z”= (“r30, r31”).

Пример заполнения секции:

Working registers:			
Name	Index	Format	Commentary
rr0	0	Int16	двухбайтный рабочий регистр (r0, r1)
rr2		Int16	двухбайтный рабочий регистр (r2, r3)
rrrr0	0	Int32	четырёхбайтный рабочий регистр (r0..r3)
rrl16	16	Int16	двухбайтный рабочий регистр (r16, r17)
ArCount			однобайтный рабочий регистр (r18)

Секция объявления имен регистров ввода-вывода

Заголовок секции: **I\O Registers:**

В этой секции предусмотрены следующие поля:

- Name – объявляемое имя.
- I/O Register – стандартное или ранее объявленное имя регистра ввода-вывода.

При объявлении допускается одному и тому же регистру давать несколько имен.

По умолчанию в среде действуют стандартные имена “p0..p63”, а также определенные в оригинальном описании микроконтроллера, такие как: “SPL”, “SPH”, “TCNT0”, “EEDR” и т.д. Кроме того, в макро-операторах возможно использование имен двойных регистров, например: “ADC”, “TCNT1”.

В большинстве случаев необходимости заполнения этой секции нет, т.к. все используемые регистры уже имеют имя по умолчанию.

Пример заполнения секции:

I/O registers:		
Name	I/O register	Commentary
InputPort	PinA	объявление PortA как InputPort
OutputPort	p\$22	объявление порта \$22 как OutputPort

Объявление имен битов

Заголовок секции: **Bits:**

В этой секции предусмотрены следующие поля:

Name – объявляемое имя;

Bit – уже существующее имя или запись, определяющая бит.

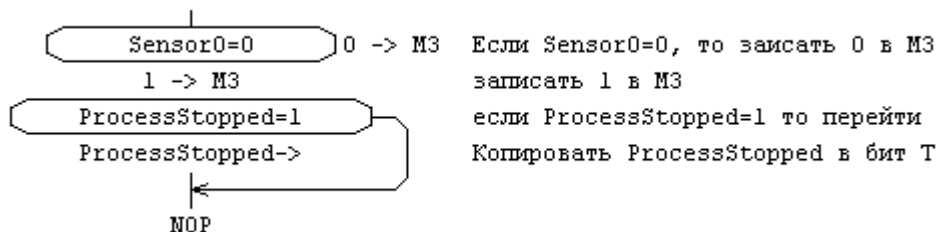
По умолчанию в среде действуют типовые имена битов регистров ввода-вывода определенные в оригинальном описании микроконтроллера, такие как: "DDA7", "ACIC", "ADEN", "ICNC1" и т.д.

Объявляемый бит может принадлежать рабочему регистру, регистру ввода-вывода, переменной SRAM или EEPROM.

Пример заполнения секции:

Bits:		
Name	Bit	Commentary
Sensor0	PinA.5	Объявление PinA.5 как "Sensor0"
M3	PortD.4	Объявление PortD.4 как "M3"
ProcessStopped	r0.4	Объявление r0.4 как "ProcessStopped"

Пример использования имен битов:



Секция объявления переменных SRAM.

Заголовок секции: **SRAM:**

В этой секции предусмотрены следующие поля:

Name – объявляемое имя переменной (ячейки памяти);

Address – (необязательный параметр) константа, определяющая конкретное значение адреса.

По умолчанию – следующий за предыдущим либо \$60 в начале компиляции;

Format – (необязательный параметр) **формат** переменной. По умолчанию принимается однобайтный формат. Многобайтные форматы используются в макро-операторах.

Count – (необязательный параметр) - число резервируемых ячеек. По умолчанию принимается равное 1.

Пример заполнения секции:

SRAM:				
Name	Address	Format	Count	Commentary
EditIndex				однобайтная переменная
ShowMode		Word		двухбайтная переменная
Reg		Int24	4	четыре трехбайтных переменных
LCD_Page			16	16 однобайтных ячеек
Phase	\$100			однобайтная переменная по адресу \$100
XArray	@Phase+4		24	24 однобайтных переменных по адресу \$104

В операторах с непосредственной адресацией SRAM “[#]->R” и “R->[#]”, имя переменной может быть использовано вместо “[#]”.

Имя переменной с префиксом “@” является константой, содержащей ее физический адрес SRAM. Приведенные ниже примеры будут откомпилированы с одинаковым результатом:

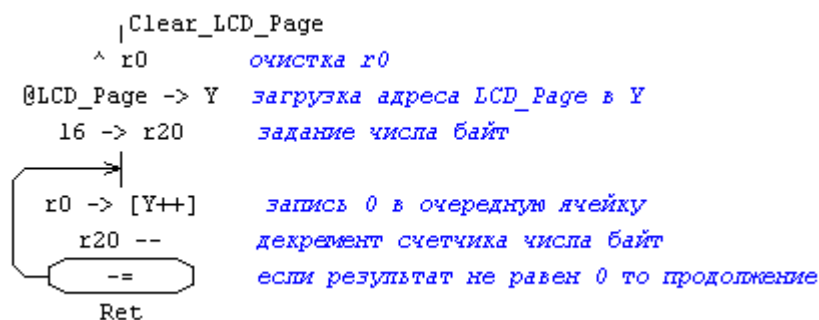
```
[$100] -> r0 , Phase -> r0 , [@Phase] -> r0
```

Объявленные многобайтные переменные могут быть использованы в макро-операторах, о которых будет изложено ниже.

Если переменная объявлена как массив (Count>1, например, “LDC_Page”), то ее имя будет указывать на первый байт массива. Для непосредственно адресации произвольного элемента массива, используйте смещение адреса. Например, если необходимо копировать r0 в пятый элемент массива, то следует записать:

```
r0->[@LCD_Page+5]
```

Приведенный ниже пример очищает все элементы массива LCD_Page:



Секция объявления переменных EEPROM

Заголовок секции: **EEPROM:**

В этой секции предусмотрены следующие поля:

Name – назначаемое имя переменной;

Address – (необязательный параметр) – константа, определяющая конкретное значение адреса.

По умолчанию – следующий после предыдущего либо 0 в начале секции;

Format – (необязательный параметр) – **формат** ячейки. По умолчанию принимается однобайтный формат. При необходимости можно задать многобайтный формат.

Count – (необязательный параметр) – число резервируемых ячеек. По умолчанию принимается равным 1.

Value – (необязательный параметр) начальные значения, представленные в виде константы или массива констант через запятую (если Count>1).

Альтернативно, начальные значения могут быть загружены из файла директивой “Load: FileName”, где FileName – имя подгружаемого файла. См. “[Непосредственное подключение файла данных](#)”.

Если данное поле не заполнено, то область переменной будет заполнена \$FF.

Пример заполнения секции:

EEPROM:					
Name	Address	Format	Count	Value	Comment
EE_Cell					однобайтная ячейка
EE_LCD_Page			16	1,2,3,4,5,6,7,8	16 однобайтных ячеек
InitValue		Word			двухбайтная ячейка
StartValue		Word	3	259,3331,0	четыре двухбайтных ячейки с загрузкой начальных значений
InitScale		DWord		#h27773F	одна четырехбайтная ячейка с загрузкой начального значения
FileTable		Word		Load: Table.db	двухбайтный массив с загрузкой из файла

В алгоритме, объявленное имя может быть использовано макро-операторах, реализующих операции с EEPROM, например:

```
InitValue -> X, 1875 -> InitValue
```

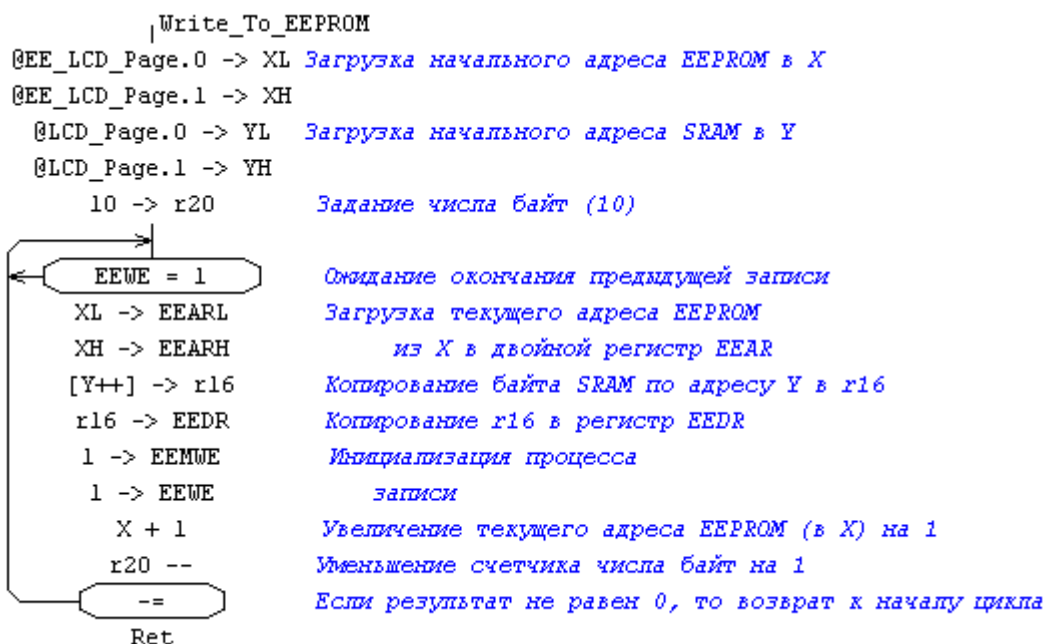
Имя переменной с префиксом "@" является константой, содержащей ее физический адрес в EEPROM. Пример копирования массива EEPROM "EE_LCD_Page" в массив SRAM "LCD_Page".

```

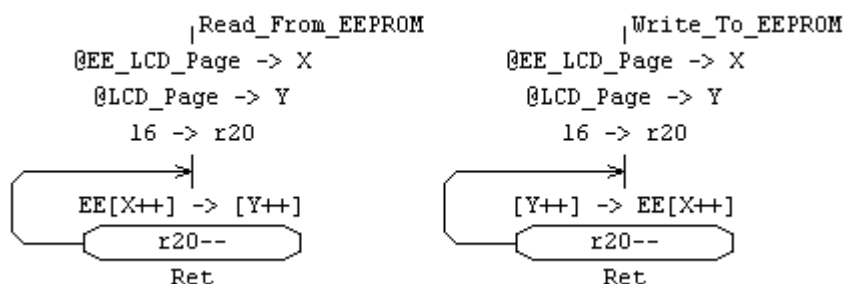
        |Read_From_EEPROM
@EE_LCD_Page.0 -> XL Загрузка начального адреса EEPROM в X
@EE_LCD_Page.1 -> XH
@LCD_Page.0 -> YL Загрузка начального адреса SRAM в Y
@LCD_Page.1 -> YH
    10 -> r20 Задание числа байт (10)
    XL -> EEARL Загрузка текущего адреса EEPROM
    XH -> EEARH из X в двойной регистр EEAR
    1 -> EERE Запись 1 в бит EERE
    EEDR -> r16 Копирование EEDR в r16
    r16 -> [Y++] Копирование r16 в SRAM по адресу в Y
    X + 1 Увеличение текущего адреса EEPROM (в X) на 1
    r20 -- Уменьшение счетчика числа байт на 1
    -- Если результат не равен 0, то возврат к началу цикла
    Ret

```

Пример копирования массива SRAM “LCD_Page” в массив EEPROM “EE_LCD_Page”.



С использованием макро-операторов эти примеры будут более лаконичными:



Представление констант в виде алгебраических выражений

Константа, обозначенная в шаблоне операторов как “#”, может быть представлена не только непосредственно, но и в виде алгебраического выражения.

Членами такого выражения могут быть:

- константы, представленные непосредственно;
- имена констант, объявленные в секции **Constants**;
- имена вершин и меток, которые содержат соответствующих мест в памяти программы.
- имена объявленные в секциях **SRAM**: и **EEPROM**: переменных с префиксом “@”, которые содержат соответствующие адреса памяти;
- имена объявленных в секции **Bits**: битов регистров с префиксом “@”, которые содержат номера этих битов;

В выражении могут быть использованы арифметические операции сложения: “+”, вычитания: “-“, умножения: “*” и целочисленного деления: “/”. Кроме того, возможно использование побитовых логических операций или: “!” и “&” и исключающего или: “^”. Например в операторе:

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

“@LCD_Page+ 5*3” – константа, представленная алгебраическим выражением.

Возможно построение сложных выражений с использованием круглых скобок.

Предусмотрена также функция возведения в степень. Синтаксис: $A(B)$, где A – основание степени, а B – показатель степени. Например, в микроконтроллере “AT90S8515” в регистре “TIFR” бит “OCF1A” имеет номер 6, а бит “OCF1B” – номер 5. Для этого случая операторы:

2 (@OCF1A)+2 (@OCF1B)->r16

r16->TIFR

будут эквивалентны с:

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

Эти операции очистят биты "OCF1A" и "OCF1B" в регистре "TIFR".

Кроме того, предусмотрены стандартные выражения, которые предоставляют свойства объявленных в таблице переменных в виде констант:

FormatOf (Var) – число байт переменной "Var" (SRAM, EEPROM, I/O или рабочий регистр);

CountOf (Var) – число зарезервированных ячеек для переменной "Var" (поле "Count") (SRAM или EEPROM).

Например, в представленном выше примере в таблице объявлена переменная "Reg" Format: Int24, Count: 4. Для этой переменной выражение "FormatOf (Reg)" будет эквивалентно константе 3, а выражение "CountOf (Reg)" - константе 4.

Макро-операторы

Программирование только с использованием элементарных операций микроконтроллера не всегда удобно. Если, например, необходимо загрузить константу \$1234 в двойной регистр X, то придется разбивать ее на старшую и младшую часть и записывать по очереди:

\$12 -> XH

\$34 -> XL

При этом теряется наглядность. Очевидна необходимость иметь возможность записи "\$1234->X", а компилятор должен сам разложить такой оператор на составляющие "\$12->XH" и "\$34->XL". А если константа в десятичном виде, то такая необходимость еще острее.

Кроме того, ряд необходимых действий невозможно реализовать непосредственно элементарными операциями. Например "37->r0" или "\$FF->DDRB". В обоих случаях необходимо сначала загрузить константу в какой-либо рабочий регистр из старшей половины, и лишь затем из него копировать в r0 или DDRB.

Макро-операторы предназначены для реализации подобных действий.

Algorithm Builder поддерживает следующие шаблоны макро-операторов:

Арифметико-логические макро-операторы. (Используется элемент "FIELD")

Шаблон	Комментарий
* -> *	Копирование
* + *	Арифметическое сложение
* - *	Арифметическое вычитание
* & *	Побитная операция "И"
* ! *	Побитная операция "ИЛИ"
* ^ *	Побитная операция "Исключающее ИЛИ"
^ *	Сброс (запись нуля)
* ++	Инкремент
* --	Декремент
- * -	Побитная инверсия
- *	Арифметическая инверсия
* >>	Логический сдвиг вправо
> * >>	Логический сдвиг вправо с переносом
± * >>	Арифметический сдвиг вправо
<< *	Логический сдвиг влево
<< * <	Логический сдвиг влево с переносом
* ->	Копирование в стек
-> *	Копирование из стека
# -> * . #	Запись 0 или 1 в бит
* . # -> * . #	Копирование бита между операндами

Макро-условия. Вписываются в элементы "CONDITION".

Шаблон	Комментарий
* = *	Переход, если равно
* != *	Переход, если не равно
* < *	Переход, если меньше
* > *	Переход, если больше
* <= *	Переход, если меньше или равно
* >= *	Переход, если больше или равно
* ±< *	Переход, если меньше с учетом знака
* ±> *	Переход, если больше с учетом знака
* ±<= *	Переход, если меньше или равно с учетом знака
* ±>= *	Переход, если больше или равно с учетом знака
* --	Декремент и переход, если результат не равен нулю
*.# = #	Переход, если бит операнда равен 0 или 1

звездочкой (*) обозначены произвольные операнды:

Операнд	Комментарий
R	Рабочий регистр (...)
P	Регистр ввода-вывода (...)
#	Константа (...)
(SRAM)	Объявленная переменная SRAM (...)
(EEPROM)	Объявленная переменная EEPROM (...)
[#]	Ячейка SRAM, адресуемая непосредственно
[X]	Ячейка SRAM, адресуемая косвенно, по X
[--X]	Ячейка SRAM, адресуемая косвенно, по X с пре-декрементом
[X++]	Ячейка SRAM, адресуемая косвенно, по X с пост-инкрементом
[Y]	Ячейка SRAM, адресуемая косвенно, по Y
[--Y]	Ячейка SRAM, адресуемая косвенно, по Y с пре-декрементом
[Y++]	Ячейка SRAM, адресуемая косвенно, по Y с пост-инкрементом
[Y+#]	Ячейка SRAM, адресуемая косвенно, по Y со смещением адреса на # байт
[Z]	Ячейка SRAM, адресуемая косвенно, по Z
[--Z]	Ячейка SRAM, адресуемая косвенно, по Z с пре-декрементом
[Z++]	Ячейка SRAM, адресуемая косвенно, по Z с пост-инкрементом
[Z+#]	Ячейка SRAM, адресуемая косвенно, по Z со смещением адреса на # байт
EE[#]	Ячейка EEPROM адресуемая непосредственно
EE[X]	Ячейка EEPROM, адресуемая косвенно, по X
EE[X++]	Ячейка EEPROM, адресуемая косвенно, по X с пост-инкрементом
EE[Y]	Ячейка EEPROM, адресуемая косвенно, по Y
EE[Y++]	Ячейка EEPROM, адресуемая косвенно, по Y с пост-инкрементом
EE[Z]	Ячейка EEPROM, адресуемая косвенно, по Z
EE[Z++]	Ячейка EEPROM, адресуемая косвенно, по Z с пост-инкрементом
LPM[Z]	Ячейка FLASH, адресуемая косвенно, по Z
LPM[Z++]	Ячейка FLASH, адресуемая косвенно, по Z с пост-инкрементом
ELPM[Z]	Ячейка FLASH, адресуемая косвенно, по Z и RAMPZ
ELPM[Z++]	Ячейка FLASH, адресуемая косвенно, по Z и RAMPZ с пост-инкрементом

Операнды, отмеченные как (...) могут быть многобайтными. Допускается подстановка операндов разного формата. При этом, если операнд, принимающий результат операции короче другого, то размерность операции будет ограничена наименьшим, в противном случае, в недостающие байты будут заполнены нулями. Например, макро-операции:

```
r16 -> Z           Z -> r16
```

преобразуются в:

```
r16 -> ZL           ZL -> r16
0 -> ZH
```

Для корректности операций, со знакопеременными величинами, у обоих операндов должен быть одинаковый формат, в противном случае, отрицательное число может быть искажено. А в макро-условиях формат операндов должен быть только одинаковый.

Операнды с косвенной адресацией являются однобайтными. Для многобайтных операций с косвенной адресацией необходимо приведение их формата. Для этого к записи необходимо добавить двоеточие и объявление формата, например:

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

такой макро-оператор будет преобразован в следующую последовательность операторов:

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

В многобайтных макро-операторах, кроме сдвигов вправо, действия начинаются с младшего байта (в операторах сдвига вправо – со старшего). Учитывая это, существуют ограничения на возможность создание макро-операций с косвенной адресацией:

- для операций, которые начинаются с младшего байта невозможно использование операндов с пре-декрементом, например: `[--X]:Word + 24000`

- для операций, которые начинаются со старшего байта (сдвиги вправо) невозможно использование операндов с пост-инкрементом, например: `[Z++]:Int24>>`

Кроме того, невозможно создание многобайтного макро-оператора с операндом `[X]`, поскольку для регистра `X` в системе команд AVR отсутствуют косвенная адресация со смещением (`[X+#]`). Используйте `[X++]` или `[--X]`.

Если создание макро-операции окажется невозможным, то компилятор выдаст сообщение: “Such macro-operation can not be created” (“Такая макро-операция не может быть реализована”).

При использовании операндов, принадлежащих к EEPROM, компилятор автоматически загрузит необходимый код, обеспечивающий чтение и запись в нее. При этом следует иметь в виду, что если производится запись в EEPROM, то в код включается ожидание окончания записи. При работе это потребует несколько миллисекунд. Конструкция макро-оператора записи в EEPROM может быть определена посредством соответствующих опций (пункт меню “Options\Project options” – “EE Macro”).

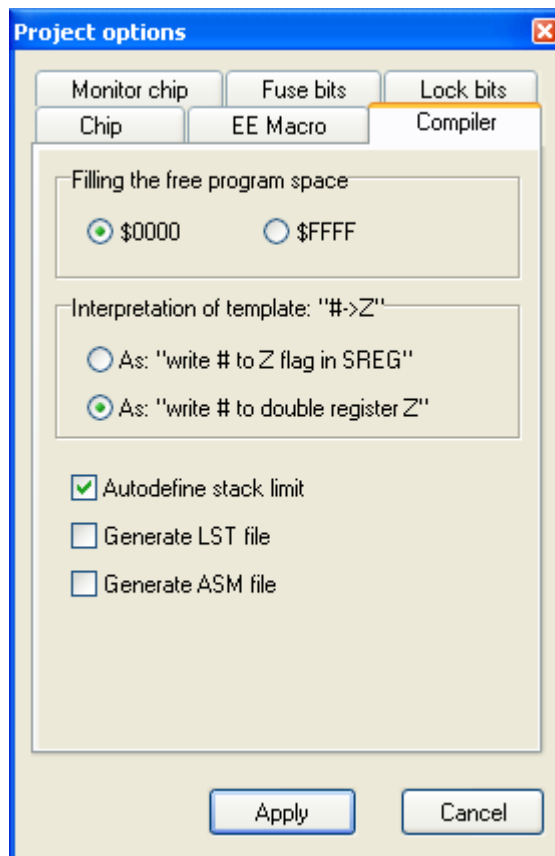
При компиляции макро-операторы преобразуются в набор элементарных инструкций микроконтроллера. При этом для реализации такого набора, как правило, используются регистры-посредники `r16` и `r17`. Поэтому, при использовании макро-операторов, во избежание недоразумений, эти регистры использовать не рекомендуется и в подпрограммах обработки прерываний целесообразно их содержимое предварительно сохранять в стеке.

Если в макро-операторе предполагается операция с константой, то в этом случае целесообразнее использовать рабочие регистры, объявленные в старшей половине. В противном случае будут использоваться промежуточные инструкции загрузки частей константы в регистр `r16`. Это может привести к увеличению размера кода, вплоть до удвоения, хотя ошибочной такая запись не будет.

Следует иметь в виду, что возбуждаемые макро-операциями флаги неадекватны аналогичным операциям на уровне элементарных операций микроконтроллера и их использование для условий переходов и переносов будет некорректно.

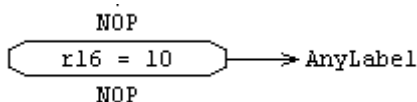
Для реализации циклов удобно использовать макро-условие: “*--”. Оно включает в себя декремент операнда и ветвление, если результат не равен нулю.

Замечание: шаблон “#->Z” может быть интерпретирован или как: “записать # в бит Z регистра SREG”, или как: “записать # в двойной регистр Z”. Для разрешения этой дилеммы, используйте опцию проекта:

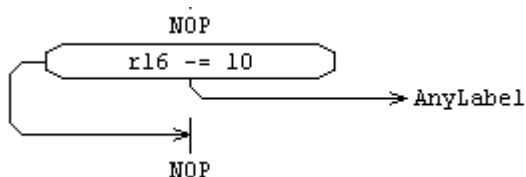


Однако, если оператор выделить жирным шрифтом, нажав клавишу “F2”, то он будет интерпретироваться только как: “записать # в двойной регистр Z”, а шаблон “#->.Z”, будет интерпретирован исключительно как: “записать # в бит Z регистра SREG”, независимо от опции.

Номинально, длина условного перехода в элементарных операторах AVR ограничена ±63 пункта программы. Если реальная длина перехода превысит этот предел, компилятор автоматически применит более сложную конструкцию с использованием инверсного условия и безусловного перехода. Например, условный переход:



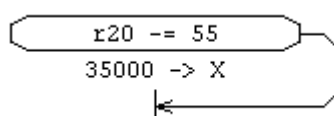
в этом случае будет откомпилирован как:



Использование макро-операторов не ухудшает эффективность кода, поскольку в них делается то, что Вам все равно пришлось бы проделать элементарными операциями.

Условные операторы

Часто в зависимости от некоего условия необходимо выполнить только **один** оператор, например:



Здесь если `r20=55`, то `35000` загружается в `X`.

В этих случаях такие действия удобнее реализовывать с помощью условного оператора. При этом, в контур должно быть вписано инверсное условие, оператор помещен как имя вектора, а сам вектор должен отсутствовать:

```
┌───────────┐ 35000 -> X  
│ r20 = 55  │  
└───────────┘
```

Метки обслуживания прерываний

Для удобства программирования Algorithm Builder поддерживает специальный вид меток – метки обслуживания прерываний. Для обслуживания прерывания обычным путем необходимо размещение по адресу вектора прерывания кода безусловного перехода на соответствующую подпрограмму. При использовании специального вида меток компилятор проделывает все это автоматически. Для этого вам необходимо дать метке (вершине) стандартное имя прерывания, и пометить ее как макро-образование, нажав клавишу “F2”, при этом имя будет отображаться жирным шрифтом. Тот же результат можно проще получить, выбрав пункт меню “Elements\Interrupt vectors\...”.

Встретив хотя бы одну такую метку в алгоритме, компилятор заполнит свободное пространство векторов прерывания кодом возврата из подпрограммы обслуживания прерывания (“**RETI**”), а по соответствующему прерыванию адресу поместит код безусловного перехода на данную метку.

Внимание: если Вы используете метки обслуживания, то начальные адреса программы будут автоматически заняты безусловными переходами на подпрограммы обслуживания прерываний. Поэтому, для того чтобы программа могла нормально стартовать, началом алгоритма обязательно должна быть макро-метка “**Reset**”. Это обеспечит загрузку в нулевой адрес безусловного перехода на начало алгоритма.

Пример:

Reset	Timer_1_Overflow
SP	Input_OVF1
Timer 1	Test_Digits
TIMSK	Show_Frequency
Init_Show	RetI
Init_Input	
✓ 1 -> I	

Если алгоритм построен таким образом, что может прервать исполнение какого-либо процесса, то в подпрограмме обработки прерываний необходимо позаботиться о сохранении состояния регистров, бит, переменных и т.д., которые могут быть искажены этой подпрограммой.

Таким образом, для того, чтобы создать прерывание, необходимо обеспечить следующее:

1. Создать вершину “**Reset**”, с которой будет начинаться исполнение программы.
2. Определить указатель стека настройщиком “**SP**” (обычно это максимальный адрес SRAM).
3. Разрешить данное прерывание. (Для таймеров – это соответствующие биты регистра TIMSK).
4. Разрешить глобальное прерывание оператором “**1 -> I**”.
5. Ввести подпрограмму обработки прерывания, которая должна начинаться с вершины с именем прерывания, а заканчиваться *обязательно* оператором “**RetI**”.

Для обслуживания прерываний загрузочной секции используйте имена прерываний с префиксом “**BOOT_**”.

Непосредственное размещение данных в памяти программы

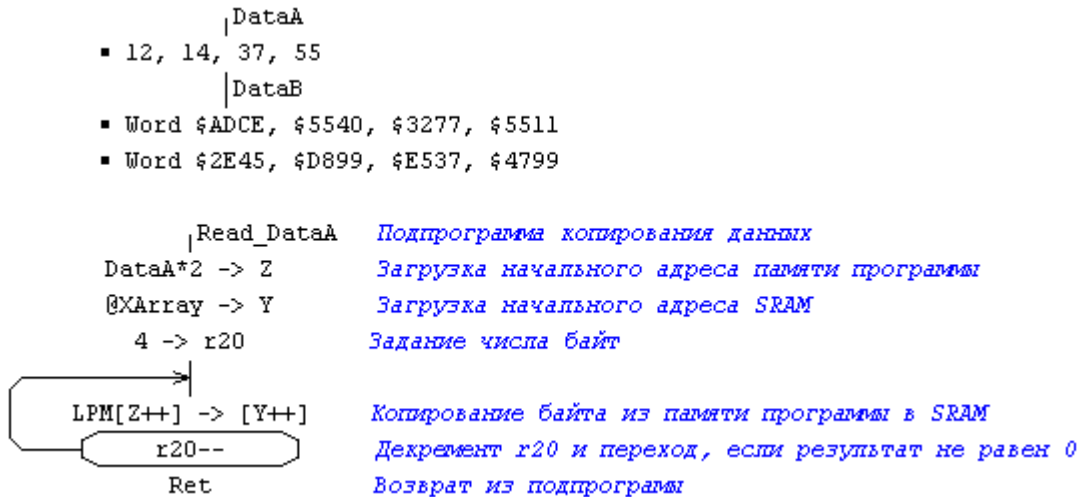
В тело программы кроме собственно программы можно поместить произвольные данные: таблицы кодов, строки сообщений и пр. Данные следует размещать после тупиковых операторов программы: “**RET**”, “**RETI**” или безусловных переходов. Для их записи используется элемент алгоритма “**FIELD**”. Для перевода этого элемента в режим непосредственного представления данных, нажмите клавиши “**Shift+F3**” или выберите пункт меню “Elements\Data”, при этом в левой части элемента появится небольшой квадрат, а строка будет привязана к левому краю. Для определения их расположения в адресном пространстве, перед ними следует поставить вершину или метку.

Синтаксис записи данных:

[Format] #, #, #..

Формат данных указывать необязательно, при этом по умолчанию будет приниматься однобайтный формат.

Например:



При чтении данных посредством операции “LPM” в регистр Z, необходимо загружать **удвоенный** адрес программы, поскольку эта операция предполагает побайтную организацию памяти программы.

Следует иметь в виду, что компилятор округляет число байт в строке данных до четного, при этом, если число байт нечетное, то последний байт будет заполнен нулем.

Данные могут быть размещены в строковом виде. Но при этом формат должен быть только однобайтным. Размещаемая строка записывается в двойных кавычках. Например:

“Hello! “

В этом случае в память программы будут помещены ANSI коды символов.

В некоторых случаях разрабатываемое Вами устройство работает с кодами, которые отличаются от ANSI. В этом случае коды символов могут быть модифицированы посредством декодирующего файла, имя которого указывается после завершающей кавычки в круглых скобках. В комплект Algorithm Builder включен файл LCD_CYR.dcd, который преобразует коды символов в коды буквенно-цифрового ЖК-дисплея с кириллицей. Например:

“ПРОЦЕСС”(LCD_CYR)

Допускается смешанное представление данных. Например:

“HELLO! “, \$0, \$DB

Непосредственное подключение файла данных

Algorithm Builder позволяет непосредственно подключить файл с данными как в тело программы, так и в качестве исходных значений EEPROM. Для этого используйте директиву: “Load: FileName”, где FileName – имя подгружаемого файла. При этом файл может иметь один из четырех форматов:

- IntelHEX (расширение “.hex”);

- General (расширение “.rom”);

- Binary (расширение “.bin”);

- формат данных Algorithm Builder (любое другое расширение, например: “.db”). Это текстовый файл, в котором записываются данные в соответствии с выше описанным форматом непосредственного размещения данных в памяти программы.

Например:


\$11, \$22, \$33

\$44, \$55, \$66, \$77, "HELLO!"
Word \$ABDC, \$FEDC

Подключение к проекту алгоритмов из других файлов

При создании нового проекта, формируется первый ведущий файл с расширением ".alp". Этот файл всегда отображается на крайней левой закладке редактора. Проект можно ограничить только одним этим файлом. Но гораздо удобнее разбить проект на несколько файлов с фрагментами общего алгоритма. Это позволяет группировать фрагменты по их функциональному назначению. Кроме того, появляется возможность использования ранее отработанных алгоритмов.

Для добавления нового файла выберите пункт меню "FileNew". При этом в окне появится новая закладка.

Algorithm Builder требует, чтобы все файлы проекта находились в одной директории с ведущим файлом "*.alp". Поэтому, для того чтобы добавить в проект уже отработанный алгоритм, сначала скопируйте его в эту директорию. А чтобы его раскрыть в редакторе, выберите пункт меню "File\Open..." или нажмите кнопку  на панели инструментов.

Если тот или иной файл раскрыт в редакторе, то это не означает, что он подключен к проекту. Раскрыть можно и файл, который не имеет к нему отношения.

Для того, чтобы подключить файл к проекту, используйте директиву компилятора "+: FileName", где FileName – имя файла (без кавычек). Для ее записи можно использовать элементы "TEXT" или "FIELD". Встретив такую директиву, компилятор приостановит свою работу в данном файле, и перейдет на указанную.

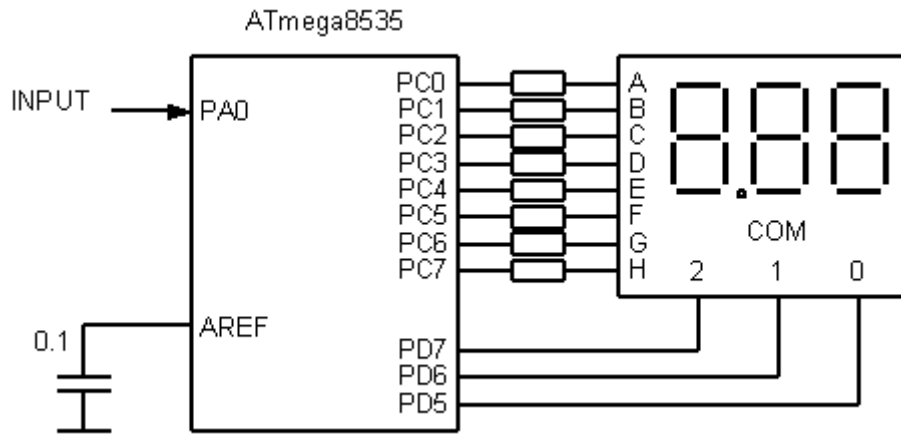
В прилагаемом ниже примере частотомера проект разбит на три файла: "Voltmeter.alp", "Display.alg" и "Arithmetic.alg". В соответствии с расположением директив "+" этого проекта, в результате компиляции получится следующая компоновка памяти программы:



Манипулируя расположением директив, можно произвольно компоновать память программы.

Пример вольтметра

Данный пример содержит более полный набор возможностей программирования в Algorithm Builder.



Вольтметр измеряет напряжение от 0 до 2 вольт с разрешением 10 мВ и отображает измеренное значение на трехсегментном светодиодном дисплее с общим катодом в режиме динамической индикации.

Алгоритм этого вольтметра находится в директории "EXAMPLES\VOLTMETER".

Проект содержит три файла: "Voltmeter.alp" – стартовый проектный файл, "Display.alg" и "Arithmetic.alg".

Генератором тактовой частоты является установленный по умолчанию внутренний RC генератор с частотой 1 МГц (Fuse bits CKSEL = 0001).

Исполнение алгоритма начинается с перехода на метку "Reset" на странице "Voltmeter". Затем настройщик "SP" загружает конечный адрес SRAM \$25F в указатель стека для обеспечения возможности вызова подпрограмм. Настройка "Timer 0" задает тактовую частоту таймера-счетчика 0 как СК/8, что обеспечивает период переполнения 2.048 мс. Настройка "TIMSK" разрешает прерывание по переполнению таймера 0.

Настройка "ADC" настраивает делитель на СК/8, что обеспечивает тактовую частоту АЦП 125 кГц (норма – 50...200 кГц), устанавливает мультиплексор входа на ADC0 (PA0), задает внутренне опорное напряжение 2.56 В, разрешает прерывание по окончании преобразования и задает циклический старт преобразования по переполнению таймера 0.

Затем вызывается подпрограмма "Init_Display", расположенная на странице "Display". Эта подпрограмма настраивает на выход порт С полностью и биты 5,6,7 порта D, сбрасывает номер отображаемого сегмента (переменная "DigitIndex") и очищает отображаемые значения цифр (трехбайтный массив "Digits").

Оператор "1 -> I" разрешает глобальное прерывание и исполнение закидывается. Далее алгоритм работает исключительно по прерываниям.

По переполнению таймера 0, с периодом 2.048 мс, аппаратно, вызывается подпрограмма "Timer_0_Overflow", в которой вызывается подпрограмма "ShowNextDigit" на странице "Display". Под меткой "DigitCodes" расположен массив семисегментных кодов цифр от 0 до 9. Подпрограмма "ShowNextDigit" каждые 2.048 мс, циклически, изменяет индекс светящегося индикатора (переменная "DigitIndex") от 0 до 2, из массива "Digits" извлекает очередную цифру, конвертирует ее в семисегментный код и отправляет обновленное значение в порт С. Одновременно, изменяет соответствующий активный катод дисплея, перенося логический 0 на очередной бит из PD5-PD7. Таким образом, приблизительно каждые 2 мс на дисплее загорается очередная цифра.

По завершении преобразования, АЦП вырабатывает прерывание, которое вызывает подпрограмму "ADC_Complete". В этой подпрограмме результат преобразования умножается на коэффициент "k_ADC" с помощью подпрограммы "fMul_XY", расположенной на странице "Arithmetic". Эта подпрограмма обеспечивает умножение содержимого двойных регистров X на Y с загрузкой результата в Z. Умножение реализует формулу: $X * Y / 65536 \rightarrow Z$. Умножение на коэффициент переводит результат преобразования в десятки милливольт. Формирование масштабирующего коэффициента "k_ADC" реализуется в таблице страницы "Voltmeter". Предварительно, вводится измеренное внешним вольтметром на выводе "AREF" значение внутреннего опорного напряжения в милливольт (константа "Vref").

Далее, подпрограмма "Z_to_Digits" извлекает из полученного числа сотни, десятки и единицы, загружая их в соответствующие ячейки массива "Digits", которые потом будут отображены на дисплее.

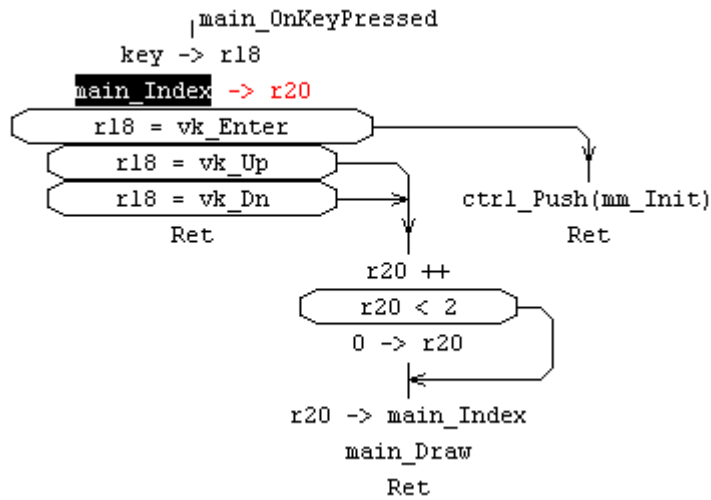
Редактирование алгоритма

Выбор редактируемого элемента делается клавишами "Up" или "Down" в порядке их размещения в памяти или левой кнопкой мыши.

Выделение фрагмента.

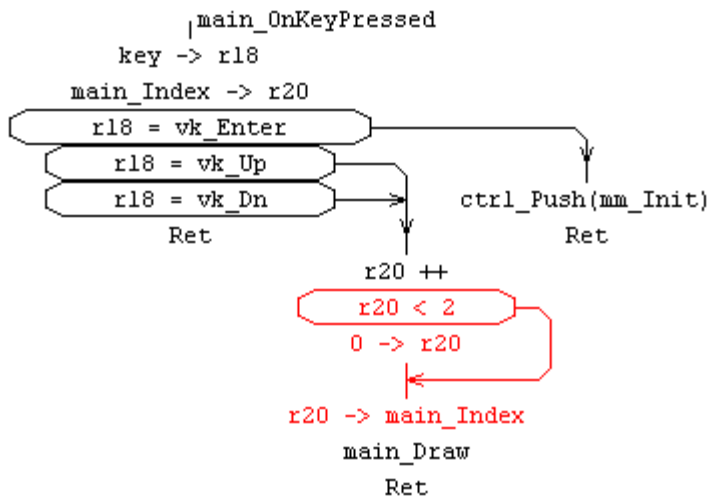
Здесь возможны три ситуации.

1. Выделение внутристрокового фрагмента.



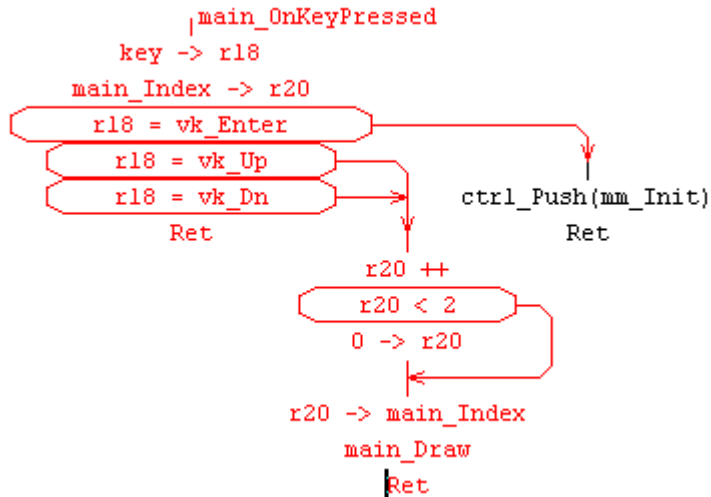
Используйте клавиши “Shift+Left” или “Shift+Right”.

2. Выделение внутриблочного фрагмента.



Используйте клавиши “Shift+Up” или “Shift+Down”. При этом среди выделяемых элементов *не должен* быть элемент “Vertex”.

3. Выделение блоков целиком.



Используйте клавиши “Shift+Up” или “Shift+Down”. В этом случае среди выделяемых элементов *должен* быть хотя бы один элемент “Vertex”. Кроме того, выделение блоков возможно с помощью окна, формируемого мышью при нажатой левой кнопке в комбинации с клавишей “Shift”. Еще, для того, чтобы выделить блоки целиком, щелкните левой кнопкой мыши в комбинации с клавишей “Ctrl”.

По умолчанию, редактируемый элемент считается выделенным внутриблочным фрагментом.

Для **копирования** выделенных фрагментов или редактируемого элемента в буфер используйте клавиши “Ctrl+C” или “Ctrl+Insert”.

Для **удаления** без сохранения в буфере используйте клавиши “Ctrl+Delete”, а с сохранением – “Ctrl+X” или “Shift+Delete”.

Для **вставки** используйте клавиши “Ctrl+V” или “Shift+Insert”. При этом возможны три ситуации.

1. Если в буфере внутристроковый фрагмент, то он вставляется внутрь строки от курсора.
2. Если в буфере внутриблочный фрагмент, то вставка производится внутрь блока ниже редактируемого объекта.

3. Если в буфере блок целиком или несколько блоков, то вначале появится контур вставляемого фрагмента. Далее мышью или клавишами направления выберите место вставки, и левой кнопкой завершите вставку. Отмена вставки производится клавишей “Escape”.

Для размещения и удаления **точки останова**, используйте клавишу “F5”.

Для перевода элемента в состояние **макро-оператора** и обратно, используйте клавишу “F2”. При этом шрифт макро-оператора будет жирным.

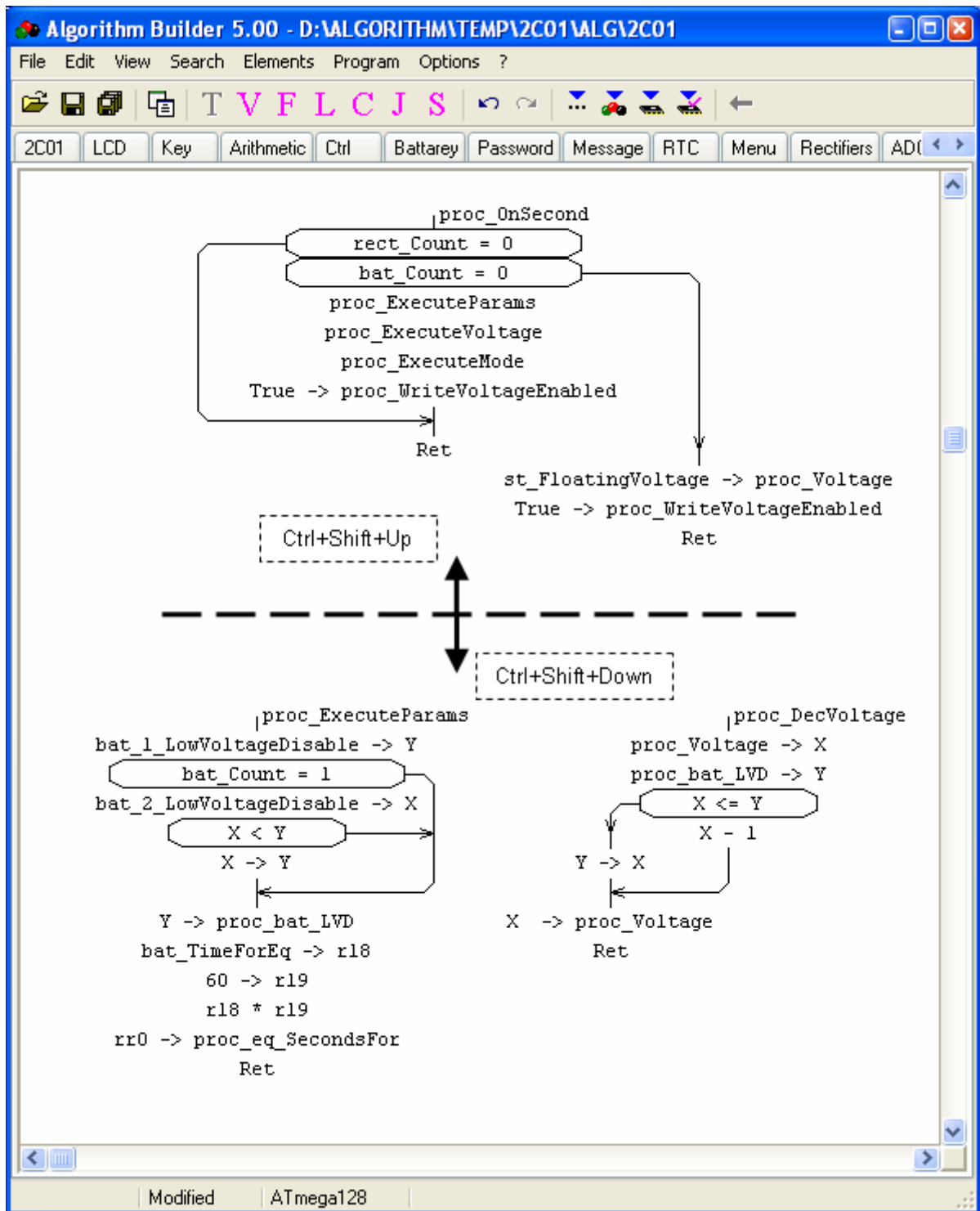
Для **прокрутки** рабочего поля алгоритма используйте линейки прокрутки или мышшь, нажав левую кнопку на свободном участке. Кроме того, для вертикальной прокрутки можно использовать колесо мыши.

Для перемещения фокуса редактора **по вершинам** блоков используйте клавиши “PgUp” и “PgDn”.


Для перемещения к **началу** алгоритма, используйте клавиши “Ctrl+PgUp”, а к **концу** – “Ctrl+PgDn”.

Для **перемещения блока**, в котором находится фокус редактора, используйте клавиши “Up”, “Down”, “Left” и “Right” в комбинации с клавишей “Ctrl” или мышшь при нажатой левой кнопке внутри блока.


Для перемещения редактируемого и всех нижеследующих блоков вверх или вниз пользуйтесь клавишами "Up" или "Down" в комбинации с клавишами "Ctrl+Shift". Это может быть необходимо для подготовки или удаления пустого пространства внутри алгоритма по вертикали:



Перемещение группы выделенных блоков производится с помощью мыши при нажатой левой кнопке, либо клавишами "Up", "Down", "Left" или "Right". Для отмены выделения и прекращения перемещения нажмите клавишу "Escape".

Для отката назад, отмены совершенных действий, используйте клавиши "Alt+BkSpace" или кнопку .

Для упорядочения блоков в памяти с целью оптимизации, вначале выделите несколько блоков. Лучше это проделать с помощью левой кнопки мыши в комбинации с клавишей "Ctrl", при этом блоки следует выделять в последовательности необходимого расположения, затем выберите пункт меню "Edit\Group selected box".

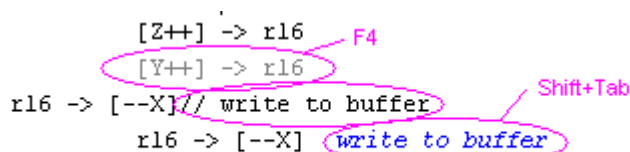
Для перехода к вершине (или метке) по ее имени, сделайте двойной щелчок на нем (переход к телу подпрограммы). Для возврата назад – нажмите кнопку  или выберите пункт меню “Edit/Back”.

Комментарии.

Создать комментарий можно несколькими способами.

1. Редактируемый элемент можно сделать пассивным нажатием клавиши “F4”. Такой элемент будет отображаться серым цветом (по умолчанию) и будет целиком игнорироваться компилятором. Повторное нажатие клавиши “F4” вернет его в активное состояние.
2. Компилятор будет также игнорировать фрагмент строки следующий за двумя косыми “//”.
3. Для ввода отдельного комментария к оператору, используйте клавиши “Shift+Tab”.

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



Редактирование таблицы распределения ресурсов

Для перемещения фокуса редактора по таблице используйте клавиши направления или клавишу “Tab” (к следующей графе) или клавиши “Shift+Tab” (к предыдущей графе) или мышь, нажав левую кнопку на выбранной ячейке.

Для **добавления** новой строки ниже редактируемой используйте клавишу “Enter”, а для добавления новой строки выше редактируемой – клавишу “Insert”.

Для **выделения** нескольких строк используйте клавиши “Shift+Up” и “Shift+Down”. По умолчанию, редактируемая строка считается выделенной.


Для **удаления** редактируемой строки или выделенных строк без сохранения в буфере используйте клавиши “Ctrl+Del”, а с сохранением – клавиши “Shift+Del” или “Ctrl+X”.

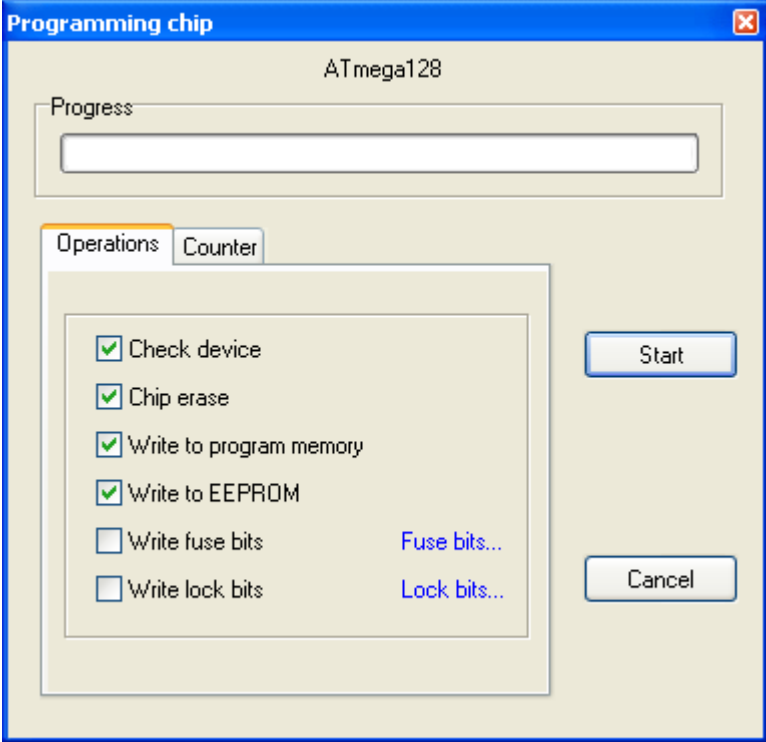
Для **копирования** выделенных строк в буфер используйте клавиши “Ctrl+Insert” или “Ctrl+C”.

Для **вставки** строк таблицы из буфера используйте клавиши “Ctrl+V” или “Shift+Insert”. При этом вставляемые строки помещаются под редактируемой. Следует иметь в виду, что вставка строк из буфера возможна *только в пределах секции*, либо в те же самые секции на других страницах.

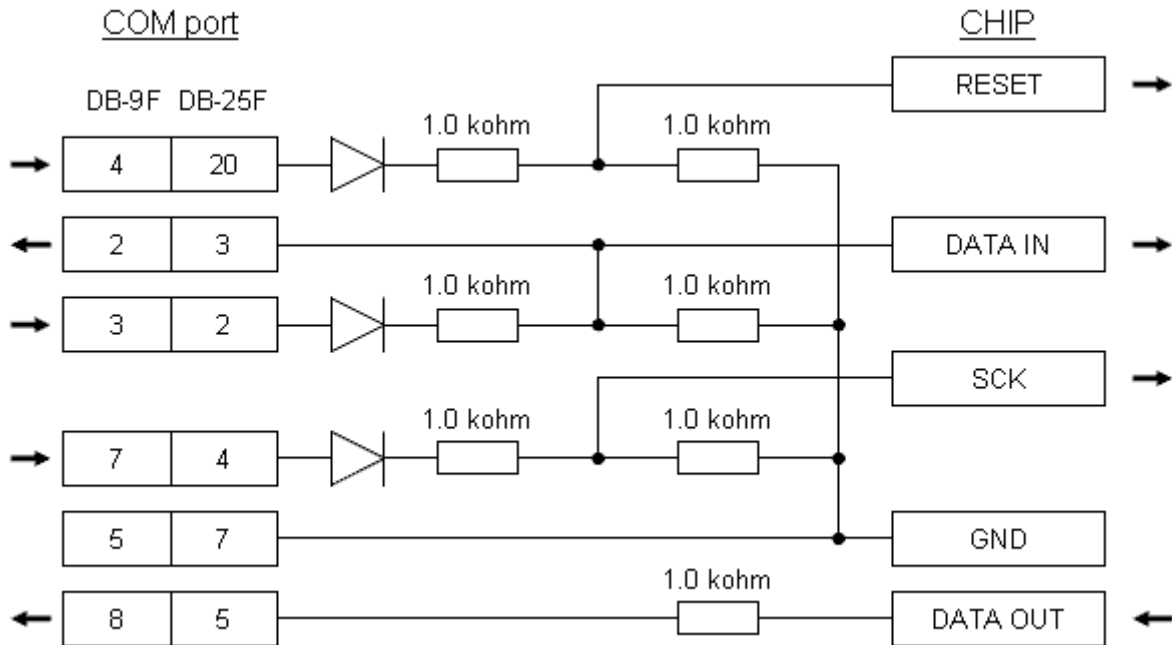
Для **изменения ширины графы** подведите курсор к ее границе возле заголовка, нажмите левую кнопку мыши и, не отпуская, переместите.

Программирование микроконтроллера

Algorithm Builder содержит встроенный внутрисхемный программатор, обеспечивающий последовательное программирование микросхем. Выбор пункта меню “Program\Run with Chip” или нажатие клавиш “Ctrl+Shift+F9” или нажатие кнопки  на панели инструментов запускает компиляцию алгоритма, и, в случае отсутствия ошибок, открывает окно программирования:



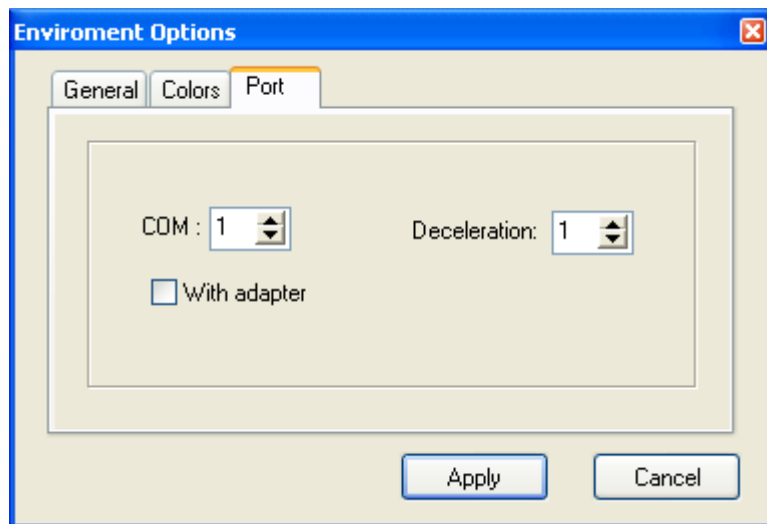
Микроконтроллер должен быть подключен к COM1 или COM2 порту через простой адаптер:



Мощность резисторов – 0.125 Вт. Диоды – любые импульсные со временем восстановления не более 50 нс (например КД522, КД510, 1N4148).

Следует иметь в виду, что данный адаптер рассчитан на стандартный COM порт с 12 вольтовыми уровнями. Однако, в ряде компьютеров, обычно ноутбуках, COM порт имеет 6 – вольтовый уровень. В этом случае номинал последовательно соединенных с диодами трех резисторов следует уменьшить до 100 Ом.

Используемый COM порт может быть выбран в опциях среды (меню: “Options\Environment Options”):



Длина кабеля, соединяющего порт с микроконтроллером, не должна превышать одного метра. При этом целесообразно использовать плоский кабель (шлейф) в котором сигнальные проводники должны чередоваться с общим (GND).

Алгоритм программирования выбирается автоматически в соответствии с типом выбранного микроконтроллера. При этом производится автоматическая проверка соответствия типа подключенного микроконтроллера выбранному.

Внешние цепи схемы не должны препятствовать сигналам с компьютера. При программировании микроконтроллера, частота подключенного к нему кварцевого резонатора должна быть не менее 1 МГц.

По завершении программирования, сигнал RESET переводится из 0 в 1, тем самым запуская загруженную программу на исполнение. Для повторного рестарта кристалла нажмите клавишу “F10”.

Среда обеспечивает подсчет числа перепрограммирования кристалла. При этом информация о количестве хранится в самой микросхеме. Для этого выделяются два старших байта EEPROM.

Для электростатической безопасности общий провод программируемого устройства следует заранее объединить с корпусом компьютера.


Lock и Fuse биты.


Настройка Lock и Fuse bits производится в опциях проекта (пункт меню "Options|Project options..."). Там же эти биты можно считать из подключенного кристалла или записать в него независимо от программатора. Однако, если включены опции "Lock bits" и "Fuse bits" в окне программатора, то эти биты будут запрограммированы при общем программировании.


Для Fuse bits состояние означает "не запрограммировано".


Внимание! При программировании Fuse bits необходимо проявлять осторожность, поскольку активизация некоторых из них может привести к невозможности дальнейшего внутрисхемного программирования (такие как "RSTDISBL", "CKSEL" и др.).



Отладка алгоритма в симуляторе


Для запуска исполнения алгоритма в симуляторе либо выберите пункт меню "Program|Run with simulator", либо нажмите клавишу "F9", либо кнопку  на панели инструментов. При этом вначале произойдет компиляция алгоритма.



Для пошагового исполнения ("Trace into") используйте либо клавишу "F7", либо кнопку .

Для пошагового исполнения без захода в подпрограммы и макро-образования ("Trace over") используйте клавишу "F8" или кнопку .

Для исполнения до выхода из текущей подпрограммы ("Trace out") используйте клавишу "F6" или кнопку .

Для запуска на исполнение алгоритма до точки останова используйте клавишу "F9" или кнопку , а для исполнения до выделенного элемента – клавишу "F4" или кнопку .

Для остановки исполнения решение, нажмите клавишу "F2" или кнопку .

Для добавления или удаления точки останова () на редактируемом элементе, нажмите клавишу "F5". Следует иметь в виду, что останов на этой точке происходит *только* по запуску клавишей "F9" или кнопкой .

Когда исполнение алгоритма остановлено, синяя метка указывает на оператор, перед которым произошла эта остановка.

Наблюдать и модифицировать текущее состояние различных компонентов микроконтроллера можно раскрыв необходимые окна через пункты меню "View|...".

Для управления состоянием окон используйте всплывающее меню, нажав правую кнопку мыши.

Для добавления наблюдаемых переменных в окнах "Watch" выберите пункт всплывающего меню "Add Watch". Для выделения размещенных переменных для последующего удаления используйте левую кнопку мыши в комбинации с клавишей "Shift" или "Ctrl". Для перемещения выделенной переменной по списку вверх/вниз, используйте клавиши "Up" и "Down", однако при этом выделена должна быть одна только эта переменная.

В табличных окнах (Watches, Maps), если в результате выполненных операций в ячейку была произведена запись, то она будет отображаться **пурпурным** цветом, а если она при этом была модифицирована, то **красным**.

Также, наблюдать значения переменных можно непосредственно в алгоритме во всплывающей подсказке, подведя курсор мыши к имени. Для редактирования - сделайте двойной щелчок на нем.

Для наблюдения за длительностью процессов предусмотрено окно "Process Time". Оно содержит четыре автономных счетчика циклов микроконтроллера. Для каждого из них предусмотрена возможность остановки процесса по достижении введенного числа. Для этого необходимо включить флажок "Enable". Если необходимо сбрасывать счетчик после останова, то необходимо включить флажок "Clear After Stop" ("Сброс после останова"). Если остановка процесса произошла по этому счетчику, то в окне появится красная надпись: "STOP".

Точки ввода-вывода

Для ввода или вывода текущих значений переменных из (или в) файл, предусмотрены соответствующие директивы алгоритма с синтаксисом:

`File:FileName -> *` (для ввода)

и

* -> File:FileName (для вывода)

где:

“ * ” – имя переменной (рабочие, I/O регистры, переменные SRAM или EEPROM;

“FileName” – имя файла.

Например, директива: ”File:Data.abd -> X” обеспечит загрузку очередного значения из файла “Data.abd” в двойной регистр “X”.

Формат файла задается расширением и может быть одним из четырех, описанных выше в разделе “Непосредственное подключение файла данных”.

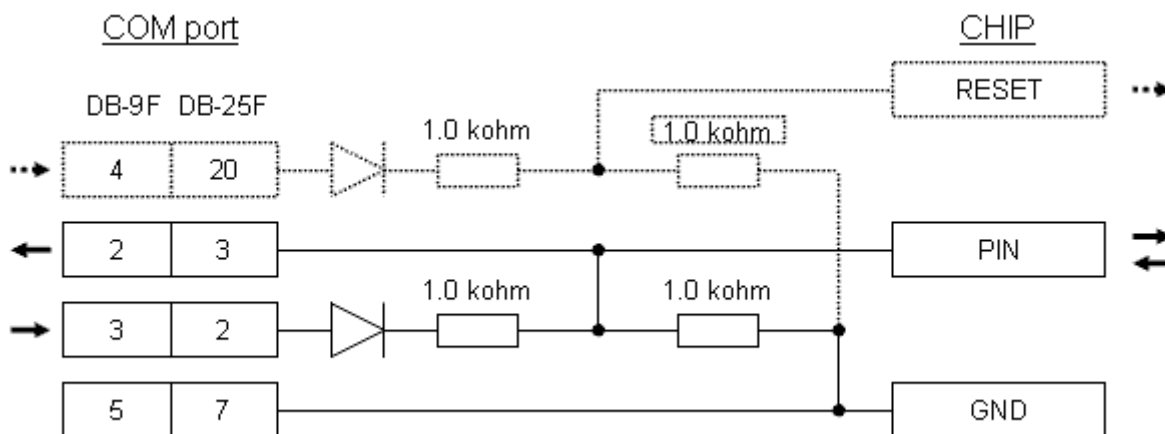
Следует иметь в виду, что эти директивы ничего не добавляют в программу, а являются только действиями симулятора, и их работа возможна только в симуляторе. Следует так же иметь в виду, что размер файла не должен превышать 256 К байт. Файл вывода создается на диске только после закрытия симулятора.

Отладка алгоритма на кристалле (мониторная отладка)

При мониторинговой отладке, компилятор добавляет к программе небольшой (160 слов) скрытый фрагмент, обеспечивающий передачу всего внутреннего состояния микроконтроллера в компьютер для последующего отображения в соответствующих окнах. При этом, состояние любого регистра, ячейки памяти SRAM или EEPROM может быть модифицировано.

Для соединения микроконтроллера с компьютером используется только один вывод, определяемый пользователем. Дополнительно может быть использована цепь сброса (RESET) для перезапуска микроконтроллера.

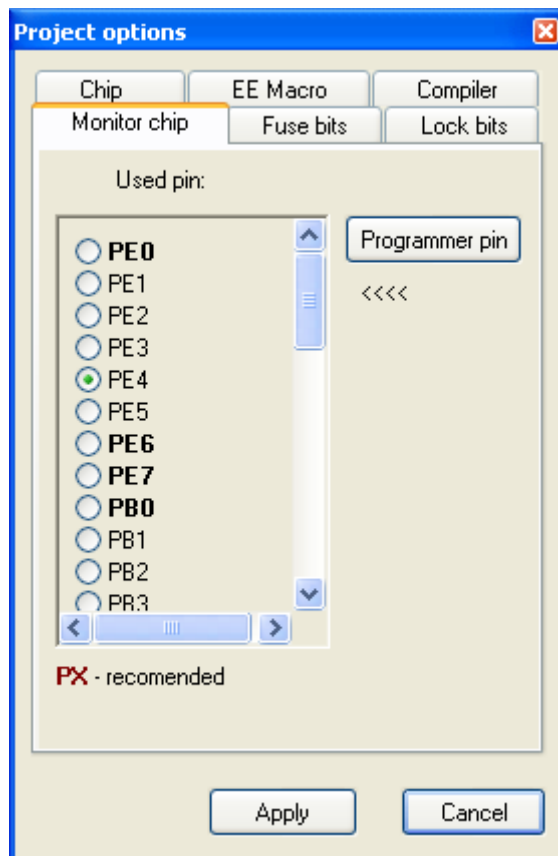
Схема адаптера для мониторинговой отладки:



Допускается использование адаптера программатора, в этом случае используемый вывод микроконтроллера будет вывод “DATA IN” (предлагаемый по умолчанию).

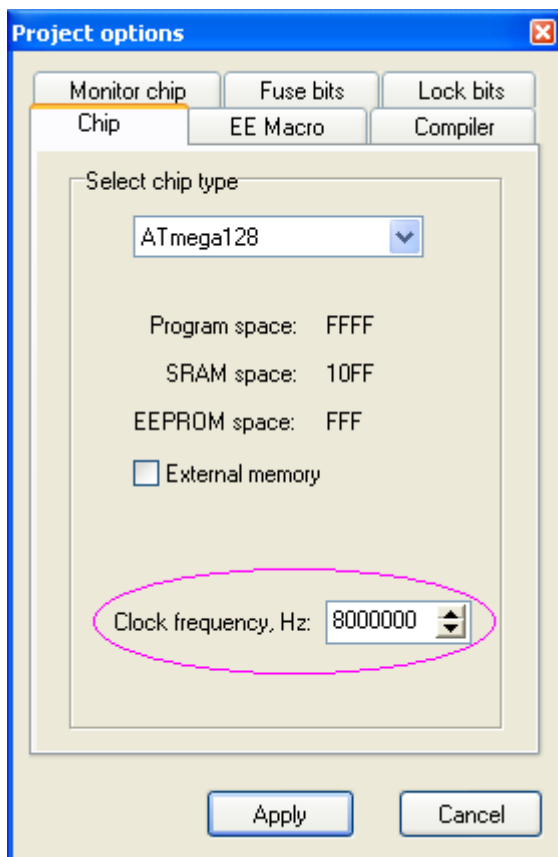
Для обеспечения работы мониторинговой отладки необходимо обеспечить следующее:

1. Определить используемый вывод микроконтроллера. Для этого необходимо раскрыть окно опций проекта (“Options/Project Options...”) на закладке “Monitor chip”:



Рекомендуемые выводы выделены жирным коричневым шрифтом. Эти выводы не имеют альтернативных порту выходных сигналов.


2. Точно указать тактовую частоту CPU:



3. К моменту первой точки останова стек должен быть уже определен.

Следует учитывать следующее:

1. Подпрограмма монитора увеличивает размер создаваемой программы на 136 слов + 1 или 2 слова на каждую точку останова (один вызов подпрограммы).
2. Монитор требует 11 байт свободного стекового пространства.
3. При попадании в точку останова исполнение программы полностью останавливается и отключается глобальное прерывание (по выходу – восстанавливается).
4. Мониторная отладка невозможна для кристаллов не содержащих SRAM.
5. Используемый вывод не должен быть занят альтернативным выходным сигналом.
6. Программа, содержащая монитор непригодна для нормального использования, поэтому по завершении отладки, микроконтроллер необходимо запрограммировать обычным образом.

Для запуска исполнения программы с отладкой на кристалле нажмите кнопку  на панели инструментов. После компиляции появится окно программатора. Запустите программирование, нажав кнопку "Start" (если программа, содержащая монитор, ранее уже была зашита, то программирование можно пропустить, нажав кнопку "Skip"). При исполнении программы до момента попадания в точку останова, окна отображения состояния микроконтроллера будут неактивными.

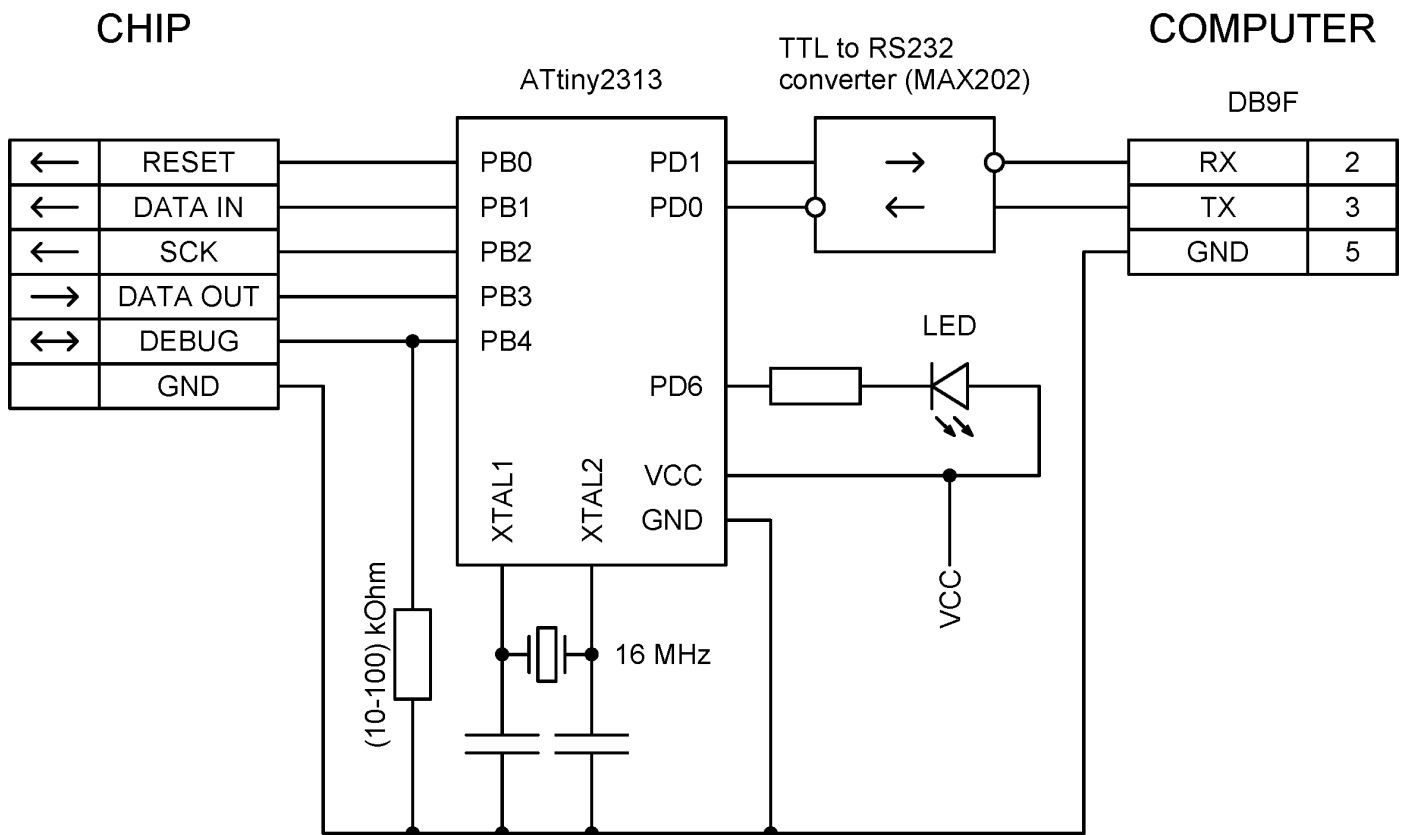
Следует обратить внимание, что обмен между компьютером и кристаллом производится асинхронным последовательным кодом и для нормальной работы требуется точное согласование частот. Поэтому мониторинг отладку рекомендуется использовать при применении кварцевого резонатора. Кроме того, тактовая частота CPU должна быть введена с учетом возможного программного изменения ее (например, регистром XDIV)

Активный адаптер для программирования и мониторинга отладки.

Представленный выше простой адаптер на резисторах и диодах обеспечивает удовлетворительную работу через стандартный COM порт. Однако при этом обеспечивается невысокая скорость программирования в силу ограничений самого COM порта при работе в таком режиме. При небольших размерах проекта это не имеет значения, поскольку программирование длится несколько секунд. Программирование большого проекта, размером более 10 Кб может длиться уже несколько минут.

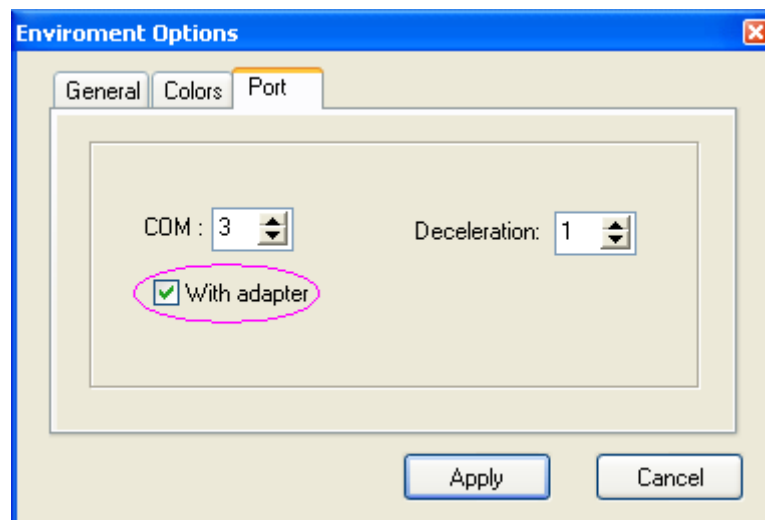
Кроме того, во многих компьютерах *нет COM порта*, а использование USB-RS232 адаптеров невозможно при таком подключении.

Представленный ниже активный адаптер обеспечивает программирование с максимально возможной скоростью, а так же позволяет использовать конвертер USB-RS232.



Программа для ATtiny2313 находится в директории "EXAMPLES\CommAdapter". Микросхема может быть запрограммирована либо через простой адаптер, представленный выше, либо с помощью любого иного программатора.

Для программирования через данный адаптер необходимо выставить соответствующий флаг в опциях среды:



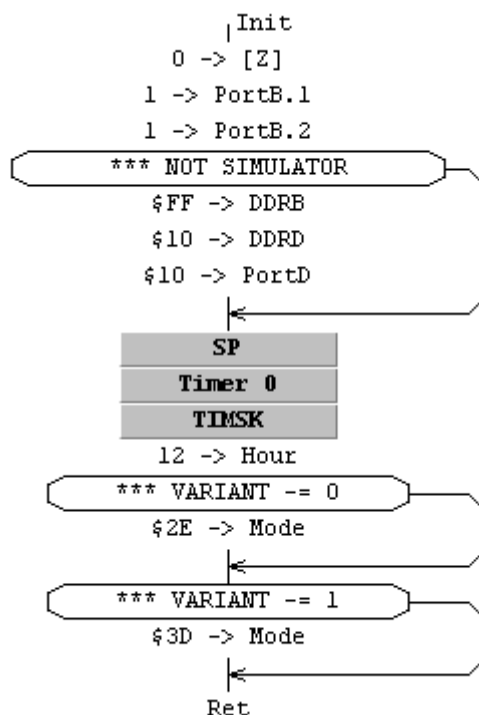
При низкой частоте CPU кристалла скорость программатора может оказаться слишком высокой. В этом случае пробуйте увеличивать замедление (Deceleration). По умолчанию, CPU большинства микроконтроллеров работает от внутреннего RC генератора 1 MHz. Для этой частоты потребуется замедление 2.

Вариант практической схемы с **оптронной развязкой** представлен в файле "EXAMPLES\CommAdapter\OpticalIsolator.pdf"

Условная компиляция.

Компилятор позволяет включать в программу фрагменты в зависимости от выполнения тех или иных условий. Иногда возникает необходимость того, чтобы, например, при отладке алгоритма в симуляторе были исключены одни фрагменты и включены другие, а при прошивке в микроконтроллер – наоборот. Также есть необходимость в выборочном включении фрагментов в зависимости от значений объявленных констант. Это позволяет использовать один и тот же файл проекта для разных вариантов применения.

Для реализации условной компиляции используется элемент "CONDITION". Вписываемое условие компиляции должно начинаться с трех звездочек : "***". В качестве условия компиляции могут быть либо зарезервированные слова "Simulator" или "Not simulator", либо алгебраическая операция сравнения двух констант. Например:



В данном примере фрагмент:

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

будет включен в программу только для симулятора;

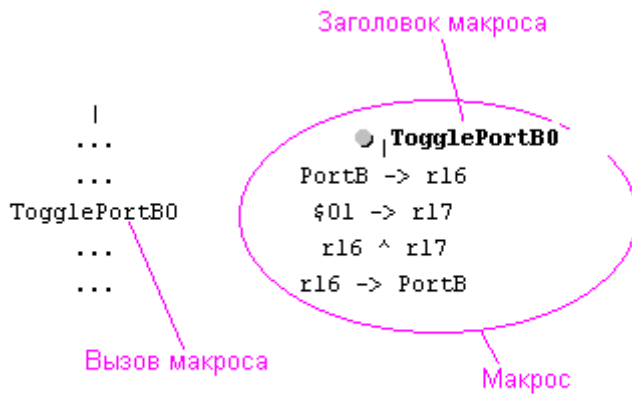
фрагмент: "\$2E -> Mode" будет добавлен только при значении 0 объявленной константы "Variant";

а фрагмент: "\$3D -> Mode" – только при значении 1.

Пользовательские макросы

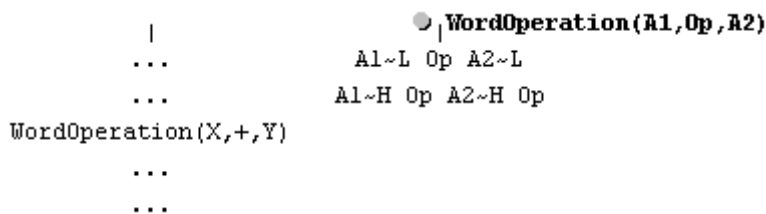
Algorithm Builder допускает создание произвольных пользовательских макросов. Тело макроса должно занимать один блок операторов. Заголовком макроса является строка вершины. Для перевода вершины в состояние вершины макроса, нажмите клавиши "Shift+F2" или выберите пункт меню "Elements/MACRO". При этом рядом со штрихом вершины появится серый кружок, и заголовок будет отображен жирным шрифтом.

Например:



Для вызова макроса в программе необходимо записать его имя в элементе “FIELD”, подобно вызову подпрограммы.

Кроме того, компилятор позволяет создание пользовательских макросов с параметрами. В этом случае параметры должны быть перечислены поле заголовка в круглых скобках через запятую. При записи операторов макроса символ “ ~ ” будет выполнять только разделительную функцию, и компилятор будет его игнорировать. При вызове такого макроса компилятор заменит объявленные в макросе параметры на вызываемые. Например:



В этом примере вместо вызова макроса:

`WordOperation(X,+,Y)`

компилятор поместит операторы:

```

XL + YL
XH + YH +

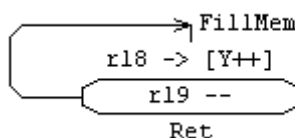
```

В случае если при компиляции внутри макроса обнаружится ошибка, компилятор укажет на место возникновения ошибки. Для того чтобы показать место вызова этого макроса выберите пункт меню “Search/Show macro call”.

Подпрограммы с параметрами

Часто подпрограммы перед вызовом установки требуют начальных значений. Чаще всего для этого используют рабочие регистры, реже – переменные SRAM. В этом случае перед операцией вызова размещаются операторы загрузки этих начальных значений. Такая форма записи не очень удобна и ненаглядна. С помощью пользовательских макросов, Algorithm Builder позволяет оформлять вызов таких подпрограмм в удобной форме, с указанием параметров в скобках, как в языках высокого уровня.

Ниже приведен пример подпрограммы FillMem, которая заполняет заданный участок памяти SRAM заданным значение.



Эта подпрограмма требует предварительно загрузить:

- в **Y** – начальный адрес участка памяти;
- в **r19** – количество заполняемых байт;
- в **r18** – заполняемое значение.

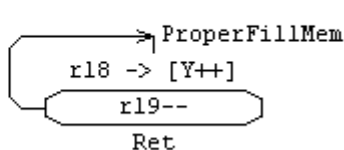
Обычным образом, вызов этой подпрограммы мог выглядеть следующим образом:

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

Такой вызов обеспечит загрузку \$22 в 16 байт SRAM, начиная с адреса \$200.

Для возможности вызова подпрограммы с параметрами, подпрограмма создается совместно с макросом. Например:

```
    * FillMem(Value, Address, Count)
    Value -> r18
    Address -> Y
    Count -> r19
    ProperFillMem
    Ret
```



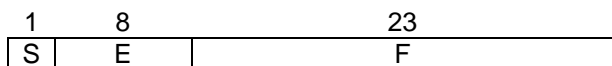
В этом случае, вызов такой подпрограммы будет выглядеть следующим образом:

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

Вместо констант параметрами могут быть рабочие регистры, переменные SRAM, EEPROM, I/O регистры и т.д.

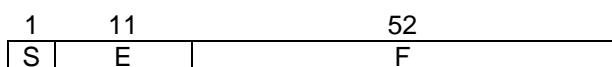
Формат данных с плавающей точкой

Algorithm Builder поддерживает работу с величинами представленными в формате с плавающей точкой. При этом такие величины могут быть представлены в 32 битном формате или в 64 битном (двойная точность). 32-битный формат:



Действительная величина V определяется как:
если $0 < E < 255$, то $V = (-1)^S * 2^{(E-127)} * (1.F)$
если $E=0$ и $F=0$, то $V = (-1)^S * 0$
если $E=255$ и $F=0$, то $V = (-1)^S * INF$ (бесконечность)
если $E=255$ и $F \neq 0$, то V is a NAN (не число)

64-битный формат:



Действительная величина V определяется как:
если $0 < E < 2047$, то $V = (-1)^S * 2^{(E-1023)} * (1.F)$
если $E=0$ и $F=0$, то $V = (-1)^S * 0$
если $E=2047$ и $F=0$, то $V = (-1)^S * INF$ (бесконечность)

если E=2047 и F=0, то V is a NAN (не число)

Представляемая в формате с плавающей точкой константа должна иметь десятичную точку либо и (или) показатель степени числа 10 после буквы “E” или “e”. Например, “1.27” или “255.99E-22” или “5e12”.

По умолчанию, константа представляется в 32-битном виде. Для представления константы в 64-битном виде к константе следует дописать преобразование: “:Int64” или “:QWord”. Например, “349.85” будет преобразовано в 32-х битное число \$43AECCD, а “349.85:Int64” – в 64-х битное \$4075DD999999999A.

В окнах “Watches” симулятора для значение переменной может быть отображено в формате с плавающей точкой только если они 32- или 64-битные.

Следует иметь в виду, что единственно корректная стандартная макрооперация для таких переменных – это копирование. Например, если объявлены 32-битные переменные SRAM “A32” и “B32”, то допустимы следующие операции:

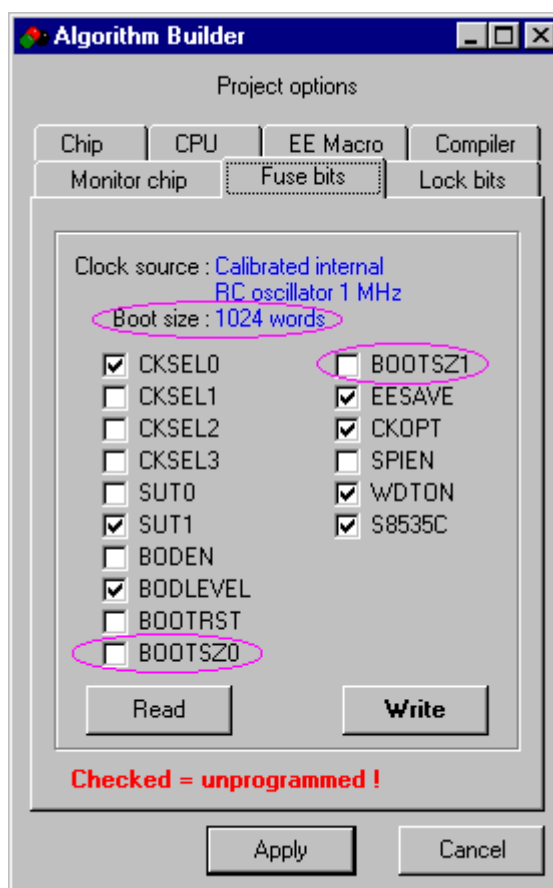
```
0.0527 -> A32          (эквивалентно: “$43AECCD -> A32”)  
A32 -> B32
```

Для осуществления математических операций, используйте соответствующие библиотеки подпрограмм, например, файл “Float32.alg”, располагающийся в директории “EXAMPLES”.

Программирование загрузчика

Для программирования загрузчика, используйте директиву “BOOT:” в элементе “TEXT”. При этом все операторы, располагающиеся ниже этой директивы, будут размещаться в загрузочной секции.

Адрес начала этой секции определяется Fuse битами BOOTSZ в опциях проекта:



Для обслуживания прерываний следует использовать имена с префиксом “BOOT_”.

При запрограммированном Fuse бите BOOTRST (=0), программа будет стартовать сразу в секции загрузчика. Для переключения прерываний в эту секцию установите бит IVSEL в 1. Для изменения этого бита, можно использовать встроенные макрооперации соответственно “SetIVSEL” и “ClearIVSEL”.

По умолчанию в Algorithm Builder объявлены следующие константы:

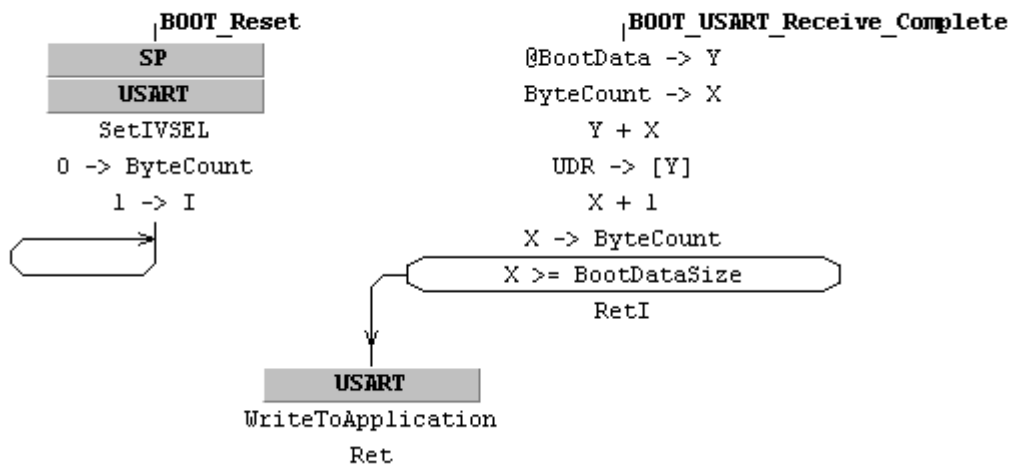
```

SPM_Load_Buffer      = #b000000001
SPM_Page_Erase      = #b000000011
SPM_Page_Write      = #b000000101
SPM_Buffer_Erase    = #b000100001
SPM_RWW_Read_Enable= #b000100001

```

Пример программы загрузчика:

 BOOT:



```

WriteToApplication
✓$0000 -> PageAddress
@BootData -> Y
*** Erasing page ***
PageAddress -> Z
SPM_Page_Erase -> SPMCR
SPM
SPMCR.0 = 1
*** Loading page ***
Flash_Page_Size -> r18
[Y++]:Word -> rr0
SPM_Load_Buffer -> SPMCR
SPM
Z + 2
r18--
*** Writing page ***
PageAddress -> Z
SPM_Page_Write -> SPMCR
SPM
SPMCR.0 = 1
Flash_Page_Size -> r18
<< r18
Z + r18
Z -> PageAddress
Z < BootDataSize
*** Set interrupt vectors ***
*** to application section ***
ClearIVSEL
*** Enable read application section ***
✓SPM_RWW_Read_Enable -> SPMCR
SPM
↓
0000

```

Полностью данный алгоритм находится в директории "EXAMPLES\BOOT RECEIVER".

Следует иметь в виду, что директива "BOOT:" действует только на текущей странице. При необходимости, для переключения компиляции в секцию приложения, используйте директиву "Application:". По умолчанию, страница компилируется в секцию приложения.

Информация о модифицированных рабочих регистрах

При разработке программы важно точно знать, какие рабочие регистры модифицируются вызываемой подпрограммой или макросом. Данную информацию можно получить во всплывающей подсказке, подведя курсор мыши к имени подпрограммы или макроса. При этом, отображается как список модифицированных регистров, так и НЕ модифицированных. Но для этого проект должен быть предварительно откомпилирован.

Если в теле подпрограммы или макроса присутствует косвенный вызов подпрограммы, то список модифицированных рабочих регистров будет не точным. В этом случае, к списку добавляется соответствующий комментарий.

Файлы проекта

Создаваемая в редакторе страница с алгоритмом сохраняется на диске с расширением “.alp” для первой, базовой страницы проекта и “*.alg” для остальных. При загрузке файла в редактор, на диске создается его копия с расширением “~al”, которой при необходимости можно воспользоваться для восстановления исходного файла. С расширением “.ini” и именем основного файла проекта (*.alp), располагающегося на самой левой странице редактора, создается файл, который содержит в себе конфигурацию среды для данного проекта. В нем сохраняются имена файлов алгоритма раскрытых на момент сохранения, а также размеры и положение основного окна и окон симулятора.

После успешной компиляции на диске создается файл с содержимым памяти программы с именем проекта и файл с содержимым EEPROM с именем файла проекта с префиксом “EE_”. Файлы могут быть созданы в бинарном формате с расширением “.bin”, в текстовом формате “Generic” с расширением “.gen” или в формате “Intel HEX” с расширением “.hex”. Для выбора формата необходимо выбрать пункт меню “Options\Environment Options...”.

Деинсталляция “Algorithm Builder”

Для деинсталляции выберите пункт меню “Options\Uninstall...”. Затем необходимо дать подтверждение/отмену в трех последующих диалоговых окнах.

При деинсталляции удаляется только данная версия продукта. Если на диске нет других версий, то Algorithm Builder удаляется полностью.