

---

**Compiler Manual**

***E-LAB AVRco***

**Pascal Multi-Tasking for Single Chips**

**Version for**

***AVR***

© Copyright 1996-2008 by E-LAB Computers



**Blaise Pascal** Mathematician 1623-1662

13-Jan-2008

---

The contents of this user guide is copyright protected by E-LAB Computers.

Autor Rolf Hofmann  
Editor Gunter Baab

# E-LAB

Mikroprozessor-Technik  
Industrie-Elektronik  
Hard + Software  
8-Bit • 16-Bit • 32-Bit

## Computers

**E-LAB Computers**  
Grombacherstr. 27  
D74906 Bad Rappenau  
Tel 07268/9124-0  
Fax 07268/9124-24  
<http://www.e-lab.de>  
[info@e-lab.de](mailto:info@e-lab.de)

### Important information

Everybody tries to write Software without bugs. The emphasis is on tries, because everybody knows that the more complex a Software is, the more likely it is to produce bugs.

We have the opinion, that this shouldn't have to be norm, and that we do not have to live with the problems and mistakes (although some Software giants think like that ☺).

If you should find any errors, we would be thankful for any information. We will try to solve any problems as quickly as possible.

It is also a normal international agreement that the software producer does not accept liability for any costs arising out of errors in software, unless otherwise agreed.

E-LAB Computers do not accept liability for costs resulting out of errors in the software. It is a condition of use of this Software you agree with these terms. If you do not agree, you are not permitted to use the software.

As we have said, before this exclusion of liability is international standard.

This user guide and the software is intellectual property from E-LAB Computers and therefore copyright protected.

This document and the software it relates to are solely for the use of the purchaser. The purchaser is not permitted to give give, sell or distribute these products. Distributing copies of these products to a third party is strictly prohibited.

We like to think that you as user of the software can make money from it and therefore also expect maintenance of the product. Illegal copies would make it impossible for us to be able to maintain this service.

As you see it is also in the interest of you, the user, to observe the copyright.

That's it the author

## Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>13</b>
1.1	<b>Every Toaster its Processor! .....</b>	<b>13</b>
<b>2</b>	<b>Overview .....</b>	<b>14</b>
2.1	<b>AVRco Versions .....</b>	<b>14</b>
2.2	<b>Manual Versions.....</b>	<b>14</b>
2.3	<b>Structure of the Documentation .....</b>	<b>14</b>
2.4	<b>Known Limitations .....</b>	<b>15</b>
<b>3</b>	<b>Basic AVRco Language Elements .....</b>	<b>16</b>
3.1	<b>Basic Symbols.....</b>	<b>16</b>
3.2	<b>Reserved Words .....</b>	<b>16</b>
3.3	<b>Standard Identifier.....</b>	<b>16</b>
3.4	<b>Delimiters.....</b>	<b>16</b>
3.5	<b>Program Lines .....</b>	<b>16</b>
<b>4</b>	<b>Language Reference .....</b>	<b>17</b>
4.1	<b>Types .....</b>	<b>17</b>
4.1.1	Standard scalar Types.....	17
4.1.2	Type Conversion .....	17
4.1.3	Variable Overlay.....	18
4.1.4	BOOLEAN.....	19
4.1.5	BIT .....	20
4.1.6	BITSET .....	20
4.1.7	BYTE .....	22
4.1.8	CHAR.....	22
4.1.9	STRING .....	22
4.1.10	ARRAY.....	25
4.1.11	TABLE.....	26
4.1.12	RECORD .....	27
4.1.12.1	WITH Statement for Access to Records.....	28
4.1.13	PROCEDURE .....	29
4.1.14	WORD.....	29
4.1.15	INT8 or ShortInt.....	29
4.1.16	INTEGER .....	29
4.1.17	POINTER .....	30
4.1.17.1	Pointer AutoIncrement.....	32
4.1.18	LONGWORD.....	33
4.1.19	WORD64 (*4*)(*P*) .....	33
4.1.20	LONGINT.....	33
4.1.21	INT64 (*4*)(*P*).....	34
4.1.22	FLOAT .....	34
4.1.23	DOUBLE (*4*)(*P*).....	34
4.1.24	ENUM .....	34
4.1.25	SEMAPHORE .....	35
4.1.26	PIPE.....	35
4.1.26.1	Pipe for ordinal Types.....	35
4.1.26.2	Pipe of Bit .....	36
4.1.26.3	Pipe for complex Types.....	36
4.1.27	SYSTIMER.....	36
4.1.28	SYSTIMER8.....	37



# AVRco Compiler-Manual

4.1.29	SYSTIMER32.....	37
4.1.30	PIDCONTROL.....	38
<b>4.2</b>	<b>Operators.....</b>	<b>39</b>
4.2.1	NOT.....	39
4.2.2	DIV.....	39
4.2.3	MOD.....	39
4.2.4	AND.....	39
4.2.5	OR.....	39
4.2.6	XOR.....	39
4.2.7	SHL.....	40
4.2.8	SHLA.....	40
4.2.9	SHR.....	40
4.2.10	SHRA.....	40
4.2.11	ROL.....	40
4.2.12	ROR.....	40
4.2.13	IN.....	41
4.2.14	+.....	41
4.2.15	-.....	41
4.2.16	/.....	41
4.2.17	*.....	41
<b>4.3</b>	<b>Pseudo Operators.....</b>	<b>42</b>
4.3.1	@.....	42
4.3.2	^.....	42
4.3.3	#.....	42
4.3.4	\$.....	42
4.3.5	%.....	42
<b>4.4</b>	<b>User Defined Language Elements.....</b>	<b>43</b>
4.4.1	Identifier.....	43
4.4.2	Numbers.....	43
4.4.3	Strings.....	43
4.4.4	Control Characters.....	44
4.4.5	Comments.....	44
<b>4.5</b>	<b>Expressions.....</b>	<b>45</b>
4.5.1	Operators.....	45
4.5.1.1	Unary Minus.....	45
4.5.1.2	Not Operator.....	45
4.5.1.3	Multiplying Operators.....	45
4.5.1.4	Adding Operators.....	46
4.5.1.5	Relational Operators.....	46
4.5.2	Function Designators.....	46
<b>4.6</b>	<b>Keywords.....</b>	<b>47</b>
4.6.1	PROGRAM.....	47
4.6.2	DEVICE.....	48
4.6.3	IMPORT.....	49
4.6.4	FROM.....	49
4.6.5	DEFINE.....	49
4.6.6	Hardware Imports within Units.....	50
4.6.7	DEFINE_USR.....	51
4.6.8	DEFINE_FUSES.....	51
4.6.9	IMPLEMENTATION.....	52
4.6.10	TYPE.....	53
4.6.11	CONST.....	53
4.6.11.1	Predefined Constants.....	54
4.6.11.2	Type Specification with Constant Declaration.....	54
4.6.11.3	Constant Load from File.....	55
4.6.11.4	Constant Located in Flash.....	55
4.6.12	STRUCTCONST.....	57

4.6.13	VAR .....	58
4.6.14	LOCKED .....	59
<b>4.7</b>	<b>Procedures and Functions.....</b>	<b>61</b>
4.7.1	PROCEDURE .....	63
4.7.2	PROCEDURE SYSTEM_INIT.....	64
4.7.3	PROCEDURE SYSTEM_MCUCR_INIT .....	65
4.7.4	FUNCTION.....	65
4.7.5	PROCESS.....	66
4.7.5.1	Define Options .....	68
4.7.6	TASK .....	68
4.7.6.1	Define Options .....	70
4.7.7	FORWARD.....	71
4.7.8	BEGIN.....	72
4.7.9	RETURN.....	72
4.7.10	END .....	73
4.7.11	ASM:.....	73
4.7.12	ASM;.....	73
4.7.13	ENDASM.....	73
<b>4.8</b>	<b>INTERRUPTs, TRAPs and EXCEPTIONs .....</b>	<b>74</b>
4.8.1	INTERRUPT.....	74
4.8.1.1	Push, Pop .....	76
4.8.1.2	PushAllRegs, PopAllRegs .....	76
4.8.1.3	External Interrupts.....	76
4.8.1.4	Interrupt Pins INT0..INTx.....	76
4.8.1.5	PinChangeInterrupts PCINT0..PCINT3 (*4*).....	77
4.8.2	TRAPs and Software Interrupts (SWI).....	77
4.8.2.1	Implementation of the Traps.....	78
4.8.3	EXCEPTIONs .....	79
4.8.3.1	Implementation.....	81
4.8.3.2	Functions .....	81
<b>4.9</b>	<b>Statements.....</b>	<b>82</b>
4.9.1	Simple Statements .....	82
4.9.2	Assignment Statement.....	82
4.9.3	Procedure Statement.....	82
4.9.4	Empty Statement.....	82
4.9.5	Structured Statement.....	83
4.9.6	Compound Statement.....	83
4.9.7	NOP Statement .....	83
4.9.8	Conditional Statements.....	83
4.9.8.1	IF Statement .....	83
4.9.8.2	GOTO Statement .....	84
4.9.8.3	CASE Statement.....	85
4.9.8.4	FOR Statement .....	87
4.9.8.5	WHILE Statement.....	88
4.9.8.6	REPEAT Statement.....	88
4.9.8.7	CONTINUE .....	88
4.9.8.8	LOOP Statement.....	89
<b>4.10</b>	<b>System Library - Standard .....</b>	<b>90</b>
4.10.1	TRUE .....	90
4.10.2	FALSE.....	90
4.10.3	PI .....	90
4.10.4	NIL .....	90
4.10.5	Type Conversion .....	90
4.10.5.1	BOOLEAN.....	90
4.10.5.2	BYTE .....	90
4.10.5.3	Int8.....	90
4.10.5.4	CHAR .....	90



# AVRco Compiler-Manual

4.10.5.5	WORD	91
4.10.5.6	INTEGER	91
4.10.5.7	LONGWORD	91
4.10.5.8	LONGINT	91
4.10.5.9	FLOAT	91
4.10.5.10	FLOATASLONG	91
4.10.5.11	LONGASFLOAT	92
4.10.5.12	POINTER	92
4.10.6	Character and String Functions	92
4.10.6.1	ORD	92
4.10.6.2	UPCASE	92
4.10.6.3	LOWCASE	92
4.10.6.4	UPPERCASE	92
4.10.6.5	LOWERCASE	93
4.10.6.6	COPY	93
4.10.6.7	STRREPLACE	93
4.10.6.8	TRIM	93
4.10.6.9	TRIMLEFT	93
4.10.6.10	TRIMRIGHT	93
4.10.6.11	PADLEFT	93
4.10.6.12	PADRIGHT	93
4.10.6.13	LENGTH	94
4.10.6.14	SETLENGTH	94
4.10.6.15	POS	94
4.10.6.16	POSN	94
4.10.6.17	APPEND	95
4.10.6.18	INSERT	95
4.10.6.19	DELETE	95
4.10.6.20	STRCLEAN	95
4.10.6.21	STRTOINT	95
4.10.6.22	HEXTOINT	96
4.10.6.23	STRTOFLOAT	96
4.10.6.24	STRTOARR	96
4.10.6.25	ARRTOSTR	96
4.10.6.26	EXTRACTFILEPATH	96
4.10.6.27	EXTRACTFILENAME	97
4.10.6.28	EXTRACTFILEEXT	97
4.10.7	Access to Parts of Variable / Constants	97
4.10.7.1	SWAP	97
4.10.7.2	SWAPLONG	97
4.10.7.3	MIRROR8	97
4.10.7.4	MIRROR16	97
4.10.7.5	MIRROR32	97
4.10.7.6	LONIBBLE	98
4.10.7.7	LO (Function)	98
4.10.7.8	LO (Assignment)	98
4.10.7.9	LOWORD (Function)	98
4.10.7.10	LOWORD (Assignment)	98
4.10.7.11	HINIBBLE	98
4.10.7.12	HI (Function)	98
4.10.7.13	HI (Assignment)	98
4.10.7.14	HIWORD (Function)	99
4.10.7.15	HIWORD (Assignment)	99
4.10.8	ABS	99
4.10.9	Negate	99
4.10.10	INC	99
4.10.11	INCTOLIM	99
4.10.12	INCTOLIMWRAP	100
4.10.13	DEC	100

4.10.14	DECTOLIM.....	100
4.10.15	DECTOLIMWRAP .....	100
4.10.16	VALUETRIMLIMIT .....	100
4.10.17	VALUEINTOLERANCE .....	101
4.10.18	VALUEINTOLERANCEP .....	101
4.10.19	VALUEINRANGE .....	101
4.10.20	MULDIVBYTE .....	101
4.10.21	MULDIVINT8.....	102
4.10.22	MULDIVINT.....	102
4.10.23	SQUAREDIVBYTE .....	102
4.10.24	SQUAREDIVINT8.....	102
4.10.25	SQUAREDIVINT.....	103
4.10.26	INTEGRATEB .....	103
4.10.27	INTEGRATEI8.....	103
4.10.28	INTEGRATEI.....	103
4.10.29	INTEGRATEW .....	103
4.10.30	Even.....	103
4.10.31	ODD.....	103
4.10.32	PARITY .....	104
4.10.33	ISPOWOFTWO .....	104
4.10.34	SIGN.....	104
4.10.35	SGN.....	104
4.10.36	PRED.....	104
4.10.37	SUCC.....	104
4.10.38	MIN .....	105
4.10.39	MAX .....	105
4.10.40	SIZEOF .....	105
4.10.41	BitCountOf.....	105
4.10.42	ADDR.....	105
<b>4.11</b>	<b>System Library - Bit Processing.....</b>	<b>106</b>
4.11.1	INCL.....	106
4.11.2	EXCL .....	107
4.11.3	TOGGLE .....	107
4.11.4	SETBIT .....	107
4.11.5	BIT .....	108
<b>4.12</b>	<b>System Library - Diverse System Functions.....</b>	<b>109</b>
4.12.1	SYSTEM_RESET .....	109
4.12.2	DELAY .....	109
4.12.2.1	mDelay.....	109
4.12.2.2	uDelay .....	109
4.12.2.3	uDelay_1.....	109
4.12.2.4	sDelay.....	109
4.12.3	SYSTIMER.....	110
4.12.3.1	SetSysTimer .....	110
4.12.3.2	SetSysTimerM.....	110
4.12.3.3	GetSysTimer .....	110
4.12.3.4	ResetSysTimer.....	110
4.12.3.5	IsSysTimerZero.....	111
4.12.4	LOWER.....	111
4.12.5	HIGHER .....	111
4.12.6	WITHIN .....	111
4.12.7	VAL .....	111
4.12.8	Block Functions .....	112
4.12.8.1	FILLBLOCK.....	112
4.12.8.2	FILLRANDOM (*4*).....	112
4.12.8.3	COPYBLOCK.....	112
4.12.8.4	COMPAREBLOCK.....	112
4.12.9	Pointer Access Outside the Linear Adress Range .....	113



# AVRco Compiler-Manual

4.12.9.1	FlashPtr .....	113
4.12.9.2	EEPromPtr .....	113
4.12.9.3	UsrDevPtr .....	113
4.12.9.4	BankDevPtr .....	113
4.12.10	FLUSHBUFFER .....	113
4.12.11	CRC Checksum .....	113
4.12.11.1	CRC CHECK .....	113
4.12.11.2	CRC STREAM .....	114
4.12.11.3	FLASH CHECKSUM .....	114
4.12.11.4	EEPROM CHECKSUM .....	115
4.12.12	RANDOM .....	117
4.12.13	RANDOMRANGE (*4*) .....	117
4.12.14	SQR, (*4*)(*P*): SQR_D .....	117
4.12.15	SQRT, (*4*)(*P*): SQRT_D .....	117
4.12.16	POW, (*4*)(*P*): POW_D .....	117
4.12.17	POW10, (*4*)(*P*): POW10_D .....	118
4.12.18	EXP, (*4*)(*P*): EXP_D .....	118
4.12.19	LogN, (*4*)(*P*): LogN_D .....	118
4.12.20	Log10, (*4*)(*P*): Log10_D .....	118
4.12.21	Trigonometrical Functions .....	119
4.12.21.1	TAN, (*4*)(*P*): TAN_D .....	119
4.12.21.2	TAND, (*4*)(*P*): TAND_D .....	119
4.12.21.3	ARCTAN, (*4*)(*P*): ARCTAN_D .....	119
4.12.21.4	SIN, (*4*)(*P*): SIN_D .....	119
4.12.21.5	SININT .....	119
4.12.21.6	SININT16 .....	119
4.12.21.7	SIND, (*4*)(*P*): SIND_D .....	120
4.12.21.8	COS, (*4*)(*P*): COS_D .....	120
4.12.21.9	COSINT .....	120
4.12.21.10	COSINT16 .....	120
4.12.21.11	COSD, (*4*)(*P*): COSD_D .....	120
4.12.21.12	DEGTORAD, (*4*)(*P*): DEGTORAD_D .....	121
4.12.21.13	RADTODEG, (*4*)(*P*): RADTODEG_D .....	121
4.12.21.14	ROTATEPNTi .....	121
4.12.22	TRUNC, (*4*)(*P*): TRUNC_D .....	121
4.12.23	ROUND, (*4*)(*P*): ROUND_D .....	121
4.12.24	FRAC, (*4*)(*P*): FRAC_D .....	122
4.12.25	INT, (*4*)(*P*): INT_D .....	122
4.12.26	GETTABLE .....	122
4.12.27	SETTABLE .....	122
4.12.28	Conversion to Strings .....	122
4.12.28.1	BYTETOSTR .....	123
4.12.28.2	INTTOSTR .....	123
4.12.28.3	LONGTOSTR .....	123
4.12.28.4	FLOATTOSTR .....	124
4.12.28.5	BOOLTOSTR .....	124
4.12.28.6	LONG64TOSTR (*4*)(*P*) .....	125
4.12.28.7	BYTETOHEX .....	125
4.12.28.8	INTTOHEX .....	125
4.12.28.9	LONGTOHEX .....	125
4.12.28.10	LONG64TOHEX (*4*)(*P*) .....	125
4.12.28.11	BYTETOBIN .....	125
4.12.28.12	INTTOBIN .....	125
4.12.29	BYTETOBIN .....	126
4.12.30	WORDTOBCD .....	126
4.12.31	BCDTOBYTE .....	126
4.12.32	PCU SI-Conversion (*P*) .....	127
4.12.32.1	Utility Functions .....	127
4.12.32.2	Temperature .....	127



4.12.32.3	Volume.....	127
4.12.32.4	Pressure .....	128
4.12.32.5	Length.....	128
4.12.32.6	Area.....	128
4.12.32.7	Weight .....	129
4.12.32.8	Energy .....	130
4.12.32.9	Integer Functions .....	130
4.12.32.10	Constants.....	130
4.12.33	Interpolation .....	131
4.12.33.1	InterPolX, InterPolY.....	131
4.12.34	Moving Average Filter.....	132
4.12.34.1	PresetAVfilter .....	132
4.12.34.2	SetAVfilter .....	132
4.12.34.3	AddAVfilter .....	132
4.12.34.4	GetAVfilter.....	132
4.12.34.5	DeclAVfilter .....	132
4.12.35	Network-Functions.....	133
4.12.35.1	Predefined Types .....	133
4.12.35.2	Converting Functions .....	133
4.12.35.3	Compare Functions.....	133
4.12.35.4	Miscellaneous Functions .....	133
<b>4.13</b>	<b>System Library - String Formatting .....</b>	<b>134</b>
4.13.1	Decimal Separator.....	134
4.13.2	WRITE .....	134
4.13.3	WRITELN.....	135
4.13.4	READ.....	135
4.13.5	READLN .....	136
<b>4.14</b>	<b>Error Handling .....</b>	<b>137</b>
4.14.1	RUNERR.....	137
4.14.2	RUNTIMEERR .....	137
4.14.3	CLEARRUNERR .....	138
<b>4.15</b>	<b>Multi-Task Functions.....</b>	<b>139</b>
4.15.1	SLEEP .....	139
4.15.2	SUSPEND.....	139
4.15.3	SUSPEND ALL.....	139
4.15.4	RESUME.....	140
4.15.5	RESUMEALL.....	140
4.15.6	PRIORITY .....	140
4.15.6.1	GetPriority.....	140
4.15.7	MAIN_PROC.....	140
4.15.8	IDLE PROCESS.....	141
4.15.8.1	On Idle Process.....	141
4.15.9	SCHEDULE.....	141
4.15.10	SCHEDULER ON/OFF .....	141
4.15.11	LOCK.....	142
4.15.12	UNLOCK.....	142
4.15.13	RESET PROCESS .....	142
4.15.14	SEMAPHORE .....	142
4.15.14.1	WAITSEMA.....	142
4.15.14.2	ProcWaitFlag .....	143
4.15.14.3	SETSEMA.....	143
4.15.14.4	INCSEMA .....	143
4.15.14.5	DECSEMA.....	143
4.15.14.6	SEMASTAT.....	143
4.15.15	PIPES .....	144
4.15.15.1	WaitPipe .....	144
4.15.15.2	PipeFlush.....	144
4.15.15.3	PipeSend .....	144



# AVRco Compiler-Manual

4.15.15.4	PipeRecv .....	144
4.15.15.5	PipeStat .....	144
4.15.15.6	PipeFull.....	145
4.15.16	PROCESS ID .....	145
4.15.16.1	ISCURPROCESS.....	145
4.15.16.2	GETCURPROCESS.....	145
4.15.16.3	GETPROCESSID.....	145
4.15.17	PROCESS STATE .....	145
4.15.18	DEVICE LOCK .....	146
4.15.18.1	SetDeviceLock.....	146
4.15.18.2	ClearDeviceLock.....	146
4.15.18.3	TestDeviceLock.....	146
4.15.18.4	WaitDeviceFree.....	147
4.15.19	Stack and Frame Usage .....	147
4.15.19.1	GETSTACKFREE .....	147
4.15.19.2	GETFRAMEFREE.....	147
4.15.19.3	CHECKSTACKVALID.....	148
4.15.19.4	CHECKFRAMEVALID .....	148
4.15.20	SCHEDULER CALL BACK.....	148
<b>4.16</b>	<b>PID-Controller.....</b>	<b>149</b>
4.16.1	pFACTOR .....	149
4.16.2	iFACTOR .....	149
4.16.3	dFACTOR .....	150
4.16.4	sFACTOR.....	150
4.16.5	NOMINAL.....	150
4.16.6	ACTUAL.....	150
4.16.7	EXECUTE .....	150
<b>4.17</b>	<b>Functions depending on HardWare.....</b>	<b>151</b>
4.17.1	PROCCLOCK.....	151
4.17.2	STACKSIZE, RAMpage.....	151
4.17.3	FRAMESIZE, RAMpage .....	151
4.17.4	TASKSTACK, RAMpage.....	152
4.17.5	TASKFRAME .....	152
4.17.6	SCHEDULER .....	152
4.17.7	SYSTICK.....	152
4.17.7.1	OnSysTick.....	153
4.17.7.2	SysTickStop.....	153
4.17.7.3	SysTickStart.....	153
4.17.7.4	SysTickRestart.....	154
4.17.7.5	SysTickDisable.....	154
4.17.7.6	SysTickEnable .....	154
4.17.7.7	SystemTime (*4*).....	154
4.17.8	ENABLEINTS.....	154
4.17.9	START_PROCESSES.....	155
4.17.10	DISABLEINTS.....	155
4.17.11	NOINTS, RESTOREINTS.....	155
4.17.12	CPUSLEEP .....	155
4.17.13	POWERSAVE .....	156
4.17.14	WATCHDOG.....	156
4.17.15	WATCHDOGSTART.....	156
4.17.16	WATCHDOGSTOP .....	156
4.17.17	WATCHDOGTRIG.....	156
4.17.18	GETWATCHDOGFLAG.....	156
4.17.19	{\$NOWATCHDOGAUTO}.....	157
4.17.20	ENABLE_JTAGPORT .....	157
4.17.21	DISABLE_JTAGPORT .....	157
<b>4.18</b>	<b>EEPROM.....</b>	<b>158</b>
4.18.1	Structured Constant.....	158

4.18.2	Variable.....	158
4.18.3	Memory Block.....	159
4.18.4	EEProm Access.....	159
<b>4.19</b>	<b>HEAP (*P*).....</b>	<b>160</b>
4.19.1	Implementation.....	160
4.19.1.1	Functions.....	160
4.19.1.2	Example.....	162
<b>4.20</b>	<b>BOOT VECTORS.....</b>	<b>164</b>
4.20.1	Implementation.....	165
4.20.2	Functions.....	165
4.20.3	Constants.....	165
4.20.4	Example Program.....	166
<b>4.21</b>	<b>BOOT TRAPS.....</b>	<b>172</b>
4.21.1	Implementation of the Boot Traps.....	172
<b>4.22</b>	<b>Inheritance.....</b>	<b>173</b>
<b>5</b>	<b>Multi-Tasking Programming.....</b>	<b>174</b>
5.1	Introduction.....	174
5.2	Principle of Operation.....	174
5.2.1	Processes and Tasks.....	175
5.2.2	Priority.....	175
5.2.2.1	Default Priorities.....	176
5.3	Optimal Multi-Tasking.....	176
<b>6</b>	<b>Optimization.....</b>	<b>177</b>
6.1	Library.....	177
6.1.1	Variable.....	177
6.1.2	Constant.....	177
6.1.3	Runtime.....	177
6.2	Highly Optimizing?.....	177
6.3	The "Merlin Optimizer".....	178
<b>7</b>	<b>Compiler Switches.....</b>	<b>179</b>
7.1	Memory Administration.....	179
7.1.1	Considerations about Memory Usage.....	182
7.2	External Memory.....	182
7.3	Include Files.....	183
7.3.1	Search Path for Include Files.....	183
7.4	Runtime Checks.....	184
7.5	Variable, Constant and Procedure Check.....	184
7.6	System Controlling.....	186
7.7	Conditional Compile.....	188
<b>8</b>	<b>Program Structure.....</b>	<b>190</b>
8.1	Program Frame.....	190
8.1.1	Order.....	190
8.2	Initializing.....	191
<b>9</b>	<b>Compiler Errors.....</b>	<b>193</b>
9.1	Error File.....	193
9.1.1	Type Mismatch.....	193



# AVRco Compiler-Manual

<b>10 Units (*P*)</b> .....	<b>194</b>
<b>10.1 Declaration and Construction of a Unit</b> .....	<b>194</b>
10.1.1 Unit-Header.....	194
10.1.2 Interface-Section.....	195
10.1.3 Implementation-Section.....	195
10.1.4 Initialization-Section.....	195
10.1.5 Finalization-Section.....	195
10.1.6 Uses-Clause.....	196
10.1.6.1 Search Path for Units.....	196
10.1.7 Info Part of a Unit.....	196
10.1.8 Hardware Imports within Units.....	197
<b>10.2 PreCompiled Units</b> .....	<b>197</b>
<b>11 Assembler</b> .....	<b>198</b>
<b>11.1 Overview</b> .....	<b>198</b>
11.1.1 ASM;.....	198
11.1.2 ENDASM;.....	198
<b>11.2 Assembler - Keywords</b> .....	<b>199</b>
11.2.1 Register.....	199
11.2.2 Assembler Directives.....	200
11.2.3 Operators for Constant Manipulation.....	201
11.2.4 Access to Pascal Constants and Variables.....	202
<b>11.3 Assembler Routines</b> .....	<b>202</b>
11.3.1 Local Variables and Assembler Access.....	202
11.3.2 Procedure Calls and System Functions.....	204
11.3.3 Function Results and Assembler.....	204
11.3.4 Function/Procedure Exit.....	204
11.3.5 Interrupt Procedures with Assembler.....	205
11.3.6 Constants and Optimization.....	205
<b>11.4 Assembler Switches</b> .....	<b>205</b>
<b>11.5 Assembler Errors</b> .....	<b>205</b>

## 1 Introduction

### 1.1 Every Toaster its Processor!

Some comments you could interpret like that. Microprocessors are more and more used for applications, you never thought they could be used for. Some things cannot be done without them.

Partly this results from the greatly reduced price of the chips, but also because of the miniaturization of them. More and more mechanical, electromechanical problems are being solved with processors. The developer has the problem to finding solutions with little money and time.

Because of the costs, the constructor has to decide in each case, which processor to use to have the ratio between cost and performance. It is nearly impossible to use one chip (e.g. 80C535 or 68332) for all applications.

The industry for semiconductors offers controllers, which have 16 pins / 8 bits (DM3.-) to more than 84 pins with 32 bits (DM80.-). The constructor also has to decide which kind of development tool he needs. The more varied the processor that are specified, the more development tools that are needed. Altogether, at a usual price between 4000,- and 10000,- DM for each tool, you have to pay some 10,000 DM, which contradicts to cost problem, so that everything, including the development, has to get cheaper. Some customers even prefer that development costs are not calculated.

With processor size 8051, 68HC11, Z80 etc. you find many tools that differ in price and performance. Unfortunately, nearly all tools are written in "C". Other languages are hard to find on the market. A developer who knows different kinds of languages, regards the value of the easy readability, the self-documentation and the easy maintenance of Pascal or similar languages (Modula-2, Oberon) highly.

Surely many arguments for or against a language can be relegated to philosophy, but the fact is, that in military or space projects, "C" is prohibited, only the Pascal related language ADA is allowed. Jobs relevant for security, for example controlling railways and plane electronics are often written in Pascal not in "C". This is surely not because of nostalgic reasons. So much to our decision to use Pascal instead of cryptic "C".

In order to find a way out of the above problem (Costs, no or weak tools, no "C"), E-LAB Computers has developed a Pascal compiler for a series of processor families. The aim was to build a tool, which has a wide performance range, but doesn't produce high cost internally. Because of this and also because of lack of resources on the target processors, we decided against some of the more complex functions. Further on there is now (in the profi version only) a linker and modular program/units.

In spite of this the tool is easily portable to other (small) processors. Non-Multi-Task versions are available for MicroChip's PIC. Multi-Task versions are available for the AVR from Atmel.

The tool always includes the IDE (Editor and so on), the Compiler and the Assembler. At least the IDE is equal to the much more expensive tools of the competitors. Only the AVR version includes a simulator as a bonus.



## 2 Overview

### 2.1 AVRco Versions

All AVRco Versions support all AVR Controllers with an internal RAM (for the stack). That means in practice the whole range.

#### **AVRco Profi Version:**

The Profi Version contains all available drivers, including very complex ones like e.g. a FAT16 file system and an extensive library for graphic LCDs.

The professional program development is furthermore assisted by the full support of Units.

#### **AVRco Standard Version:**

The Standard Version omits only the most complex drivers, and does not support units.

#### **AVRco Demo Version:**

The Demo Version supports all controllers and all drivers of the Standard Version.

The **only restriction** is the limitation of the generated code to max. 4 kByte size.

### 2.2 Manual Versions

Chapters marked with the attribute (\*P\*) are only available in **AVRco Profi Revision**.

Chapters marked with the attribute (\*4\*) are only available in **AVRco Revision 4**.

### 2.3 Structure of the Documentation

#### **..\E-Lab\DOCs\DocuCompiler.pdf:**

contains the Pascal language description and the enhancements compared with Standard Pascal

#### **..\E-Lab\DOCs\DocuStdDriver.pdf:**

contains the description of the drivers contained as well in the Standard, as in the Profi Version.

#### **..\E-Lab\DOCs\DocuProfiDriver.pdf:**

contains the description of the drivers contained only in the Profi Version.

#### **..\E-Lab\DOCs\DocuReference.pdf:**

contains a Short Reference (the the same as the online help)

#### **..\E-Lab\DOCs\DocuTools.pdf:**

contains the description of the IDE, the simulator, a tutorial etc.

#### **..\E-LABIDE\DataSheets\Release-News.txt:**

lists the enhancements in chronological order.

The enhancements are documented in the above mentioned .pdf files (DocuXXX.pdf)

#### **..\E-Lab\AVRco\Demos\ :**

contains many test and demo programs

#### **..\E-Lab\DOCs\ :**

contains the documentation and further schematics and data sheets

## 2.4 Known Limitations

- The function *Integratel* still has a problem with negative values.
- Overwriting of predefined Types, Variables, Constants, Functions and Procedures is not implemented yet.
- The order of the operators (\* AND SHR etc.) is not completely implemented yet. So please use always **parentheses** in conjunction with expressions.
- Not implemented: "**With**" constructs with Records in Records

- Strings

The following string **concat** result is not what you expect:

```
str:= str1 + str;
```

Because of the limited RAM this can not be realised.

Possible is:

```
str:= str + str1;
```

- Arrays

The construction of **Array of Array** is not supported



## 3 Basic AVRco Language Elements

### 3.1 Basic Symbols

The basic vocabulary of AVRco Pascal consists of basic symbols divided into letters, digits, and special symbols:

Letters A to Z, a to z and \_ (underscore). The underscore should not be used as the first letter of a symbol.

Digits 0123456789

Special symbols `*/=<>()[]{}|.,`

No distinction is made between upper and lower case letters. Certain operators and delimiters are formed using two special symbols:

Assignment operator: `: =`

Relational operators: `<> <= >=`

Comments: `(* and *)` may be used instead of `{ and }`.

It is also possible to use C-style comments using `//`

In addition, AVRco Pascal makes use of certain constructs to allow low level access to the CPU and it's resources. There are also many language extensions to support embedded programming.

### 3.2 Reserved Words

Reserved words are integral parts of AVRco Pascal. They cannot be redefined and must therefore not be used as user defined identifiers.

**Examples:**

ABS, AND, ARRAY, ASM, BEGIN, BREAK, CASE, CONST, CONTINUE, DIV, DO, DOWNT0, ELSE, ELSIF, END, ENDFOR, ENDCASE, ENDWHILE, ENDIF, EXIT, FOR, FORWARD, FUNCTION, GOTO, IF, IN, LABEL, MOD, NOT, OF, OR, PROCEDURE, PROGRAM, RECORD, REPEAT, ROR, ROL, SHL, SHR, STRING, THEN, TO, TYPE, UNTIL, VAR, WHILE, WITH, XOR

### 3.3 Standard Identifier

AVRco Pascal defines a number of Standard Identifiers for predefined Types, Constants, Variables, Procedures and Functions. These can also not be redefined.

**Examples:**

FALSE, TRUE, NIL, CHAR, BOOLEAN, INTEGER, BYTE, INT8, LONGINT, WORD, LONGWORD, FLOAT, POINTER, SIZEOF, ADDR, @, INC, DEC, MOVE, LENGTH, COPY, INTTOSTR, BYTETOSTR, INTTOHEX, STRTOINT, LO, HI, LOWORD, HIWORD, INSERT, DELETE, UPCASE, POS

### 3.4 Delimiters

Language elements must be separated by at least one of the following delimiters: a blank, an end of line, or a comment. Note that comments are allowed anywhere in the source. There are some restrictions.

### 3.5 Program Lines

The maximum length of a program line is 250 characters.



## 4 Language Reference

### 4.1 Types

#### 4.1.1 Standard scalar Types

A data type defines the set of values a variable may assume. Every variable in a program must be associated with one and only one data type. Although data types in the AVRco can be quite sophisticated, they are all built from simple (unstructured) types.

The basic data types of Pascal are the scalar types. Scalar types constitute a finite and linear ordered set of values.

A simple type may either be defined by the programmer (it is then called a *declared scalar* type), or be one of the *standard scalar types*: integer, int8, word, longint, longword, float, boolean, char, or byte.

Apart from the standard scalar types, Pascal supports user defined scalar types, also called declared scalar types (enumeration). The definition of a scalar type specifies, in order, all of its possible values. The values of the new type will be represented by identifiers, which will be the constants of the new type

##### *type*

```
Operator = (Plus, Minus, Multi, Divide);
```

In the above example, the type Operator is nothing else but a type byte. All following values are enumerated starting from 0. Effectively Plus is a constant with the value 0, Minus a 1, Multi a 2 and so on. The use of defined scalar types is strongly recommended as it greatly improves the readability of programs.

##### *type*

```
TDay = (Mon, Tue, Wed, Thur, Fri, Sat, Sun);
TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
TMarr = array[Jan..Dec] of TDay;
```

##### *var*

```
MonArr = TMarr;
```

```
...
```

```
MonArr[Aug] := Sun;
```

```
if MonArr[Jan] = Fri then ...
```

#### 4.1.2 Type Conversion

The compiler does not perform automatic type conversion. An allocation like *byte := word* leads to a **Type Mismatch**. But it is possible to convert most types to another by using the so called typecasting. In many cases no additional machine-code will be generated (*char := char(byte)*).

AVRco Pascal provides "Type Casting" as can be found in most Pascal compilers today. This is done by simply using the type identifier as "function" name. The single parameter in this "function call" is a variable of type A that you would like the compiler to view as a variable of type B (the type identifier used in the "function call").



# AVRco Compiler-Manual

```
var  ch : char;  
      b1 : byte;  
      w1 : word;  
      i1 : integer;  
      p1 : pointer
```

```
ch:= char(b1);  
b1:= byte(w1);  
i1:= integer(w1);  
p1:= pointer(w1);
```

```
type  
TRecordA = record  
           a,b,c : word;  
           end;  
TRecordB = record  
           X : longint;  
           Y : word;  
           end;
```

```
var  
  A : TRecordA;  
  B : TRecordB;  
  C : char;  
  D : byte;
```

```
begin  
  A:= RecordA (B);  
  RecordA (B):= A;  
  D:= byte (C);  
  C:= char (D);  
end.
```

## Tip:

In the Pascal world it's good practice that type declarations start with a capital "T". This is not a must but supports the readability of the program and helps to avoid programming errors.

### 4.1.3 Variable Overlay

Variables can be placed directly into other variables (Overlay).

But here is the problem that the second variable can be larger (memory) as the referenced variable. Because only the reference variable does a memory allocation, it is possible that a write access to the overlay destroys the subsequently placed variable at runtime.

```
var abc           : byte;  
    xyz           : byte;  
    ovr1[@abc]    : byte;           // it's ok  
    ovr2[@abc]    : word;          // problem
```

In this example a write access to the variable "ovr2" always overwrites also the variable "abc" as expected but also (unexpected) the variable "xyz".

The compiler checks in case of an overlay whether the overlay variable fits into the memory which was allocated by the referenced (overlaid) variable.

But to enable the construction below there is a compiler Switch `{$NOOVRCHECK}`. So it is possible to switch off the check for a single following declaration.

```
var
v1  : byte;
v2  : byte;
v3  : byte;
v4  : byte;

{$NOOVRCHECK}
v5[@v1] : longword;
```

Normally v5 will raise an error because a longword consumes 4 bytes but the variable v1 only offers one byte memory. By disabling the check the variable v5 occupies the memory locations of v1-v4 and overlays all four bytes, which can be legal but may not be so. The programmer is responsible for that.

Borland Pascal has another implementation for this. The AVRco also supports this type:

```
var
abc  : byte;
xyz  : word;
ovr1 : byte absolute abc;           // var abc overlaid
{$NOOVRCHECK}
ovr2 : word absolute xyz;          // var xyz overlaid
```

"absolute" is only applicable with variables in RAM and EEPROM area.

## 4.1.4 BOOLEAN

Required memory: 8bit, value: true..false

A boolean value can assume either of the logical truth values denoted by the standard identifiers *True* and *False*. These are defined such that *False* = 0; *True* > 0 (normally \$FF). A *Boolean* variable occupies one byte in memory.

true = \$FF    false = \$00

```
var  flag : boolean;
```

### Comment:

One boolean needs 1 byte as memory location. To economize memory location, it possibly makes sense to use a one byte-variable and divide it with the help of the BIT-definition into separate bits. So the required memory for 8 Booleans is reduced to 1 byte.

```
var  flag           : byte;
      flag0[@flag, 0] : bit;
      flag1[@flag, 1] : bit;
      ...

if Bit (Flag0) then ...
Incl (Flag1);
SetBit (flag0, flag1);    {copy bit}
```



## 4.1.5 BIT

Required memory: 1bit, true..false, 0..1

In order to handle the necessary bits for embedded applications the **variable bit** was introduced. This type can be used exactly if it were a boolean. In addition *INCL(bit)*, *EXCL(bit)*, *TOGGLE(bit)*, *SetBit* and “*if BIT(bit) then ..*” may be used. The declaration of a bit-variable **always** consists of two grades: First the memory location or variable in which the bit resides has to be declared. This variable must be a global var. Bit definitions into Records or Arrays are not possible. First define a global variable of type byte, word or with **REV4 also LongWord**.

```
var Leds[$05] : byte; or
    bits16    : word;
```

The type of the memory location (byte, word, longword) determines, too, if 8 or 16 bits are at disposal. After the declaration of a general variable the actual bit-declaration follows. The first parameter indicates a memory location and the second parameter indicates the corresponding bit of this memory location. Symbolic parameters can also be used.

```
const LedBit2 = 3;
var port6[6] : byte;
    Led2[@port6, Led2Bit] : bit;
```

```
if BIT (Led2) then ...
    Toggle(Led2);
```

```
SetBit (Led2, not Led2);
```

Bits can also be dynamically generated within the program.

```
Toggle (Leds, 3);           {Bit3 in 8 Bits}
Incl (bits16, 12);         {Bit12 in 16 Bits}
```

## 4.1.6 BITSET

8bit, 16bit or 32bit dependent of the underlying enumeration. 32bits only in REV4

### Basics:

With an enumeration each of the 256 possible values each used value can/must be named. With a BitSet each used bit must be named. Because of this a byte can contain upto 8 “Names“ = Bits, a word up to 16 names and a longword up to 32 names. With a BitSet each single bit can be accessed and manipulated by it's name. Also accessing in Bit-groups is then possible.

### Definition:

Before a BitSet type can be defined, there must be an enumeration type which contains the Bit names. The count of the bit names defines whether the resulting BitSet resides in a byte, word or longword. The maximum count of names or bits is 32.

### Declaration:

```
type
  TBitNames = (one, two, three, four, five, six); // enum
  TBitSet   = BitSet of TbitNames;             // build a bitset type

var
  BitSet1 : TBitSet; // build a bitset var
  BitSet2 : TBitSet; // build a bitset var
  Bb      : byte;
```

## Working with BitSets:

```
BitSet1:= BitSet2;           // copy a set into another
bb:= byte (BitSet1);        // type convert
BitSet2:= TbitSet (bb);     // type convert
```

In order to completely fill a bitset all bits of the enum type must be entered:

```
BitSet1:= [one, two, three, four, five, six];
```

It's also possible to use the enum name:

```
BitSet1:= [tEnum];
```

BitSets can be set in conjunction with **Operators**. These are + - \* /

### Addition:

```
BitSet1:= [one, three];
BitSet2:= [two, four, six];
BitSet1:= BitSet1 + BitSet2;           // unification – logical or
//BitSet1 contains now one, two, three, four and six.
```

### Subtraction:

```
BitSet1:= [one, two, three];
BitSet2:= [two, four, three];
BitSet1:= BitSet2 - BitSet1;           // difference – logical and not
//BitSet1 contains now four.
```

### Multiplication:

```
BitSet1:= [one, two, three];
BitSet2:= [two, four, five];
BitSet1:= BitSet2 * BitSet1;           // logical and
//BitSet1 contains now two.
```

### Division:

```
BitSet1:= [one, two, three];
BitSet2:= [two, four, five];
BitSet1:= BitSet2 / BitSet1;           // logical xor
//BitSet1 contains now one, three, four, five.
```

BitSets can be **compared** with the use of = <> <= >= **IN**

Compared are the binary values of the corresponding bit patterns.

```
if BitSet1 = [one, two, three] then ...
if BitSet2 <= [two, four, five] then ...
if BitSet2 in BitSet1 then ...
if [two, four, five] in BitSet1 then ...
```

## 4.1.7 BYTE

8 bit, 0..255

Bytes are whole numbers. They are limited to a range of 0 through 255.

Bytes occupy one byte in memory.

All variable declarations can be assigned to a fixed address.

Example for a fixed address:

```
var b[$10] : byte;           {memory at addr. 10hex}
```

Example for a free address allocation by the compiler:

```
var x : byte;
```

It is also possible to specify a previous declared variable as reference. So it is possible, for example, to access the two bytes of a word variable.

```
var w[$12]      : word;
      b1[@w]     : byte;   {b1 is at addr. 12hex – low byte}
      b2[@w + 1] : byte;   {b2 is at addr. 13hex – high byte}
```

## 4.1.8 CHAR

8 bit, chr(0..255)

A Char value is one character in the ASCII character set.

Characters are ordered according to their ASCII value, for example: 'A' < 'B'. The ordinal (ASCII) values of characters range from 0 to 255. A Char variable occupies one byte in memory.

Example variable of type char:

```
var c : char;
```

Example constant of type char:

```
const cd = 'D';
      Bell = ^G;   {Control G}
      LF = #10;   {Line Feed}
```

## 4.1.9 STRING

0..255 bytes, variable or constant.

AVRco Pascal offers the convenience of string types for processing of character strings i.e. sequences of characters. String types are structured types and are in many ways similar to array types. In Pascal, a string is the exact equivalent of an *array of char* and such an array may be treated as a string.

AVRco Pascal uses the FIRST character in a string as a length indicator with the remainder of the string following. This is compatible with most Pascal string implementations but differs from C, which uses a zero byte as string delimiter.

In short, the Pascal string results in a better performance since the length of a string is always known without having to scan the string. The Pascal string is also a convenient dynamic storage medium for data other than characters as there is no restriction on the value of a character that can be placed inside the string. C strings on the other hand may be of indefinite length while Pascal strings are restricted to a maximum length of 255 characters.

```
type st10 = string[10];           {stringlength = 10}
```

```
structconst {constant in Rom, at startup copied into Ram}  
  str : st10 = 'abcde';
```

```
const      {constant in Rom}  
  st = '1234' + 'R' + #7 + ^L;
```

```
var  st1 : string[8];
```

```
  st1:= st;  
  ch:= st[2];
```

The so called Length Byte is located at first position of the string ( str[0] ). This byte specifies the actual occupied length of the string. It is possible, for example, to change the length dynamically by manipulating the position 0 within the declaration ( str[ 0]:= # 5; ). The better way is to use the system function **SetLength** ( st : string; len : byte) ;

The length byte (= char!!) can also be read to determine the actual length. But here is the function **Length**(str) better and faster. Do string-manipulations only with Var and StructConst !

The definition of a string type must specify the maximum number of characters it can contain, i.e. the maximum length of strings of that type. The definition consists of the reserved word **string** followed by the maximum length enclosed in square brackets. The length is specified by a byte constant in the range 0 through 255. Notice that strings do not have a default length; the length must always be specified.

String variables occupy the defined maximum length in memory plus one byte which contains the current length of the variable. The individual characters within a string are indexed from 1 through the length of the string.

Strings are manipulated by the use of *string expressions*. String expressions consist of string constants, string variables, function designators, and operators.

The plus-sign may be used to concatenate strings. The *Concat* function available in some Pascal implementations does the same thing but is not implemented in AVRco Pascal since the + operator is often more convenient.

The result of string expressions cannot provide a string larger than 255 characters in length, also, in string assignments, the assignment target will never receive a string larger than the size of the target permits.

```
'E-LAB ' + 'Pascal ' + 'is ' + 'fun...  
'123' + ' ' + '456'  
'A ' + 'B' + 'C ' + 'D '
```

The relational operators = and <> are lower in precedence than the concatenation operator. When applied to string operands, the result is a Boolean value (*True* or *False*). When comparing two strings, single characters are compared from the left to the right according to their ASCII values.

If the strings are of different length, but equal up to and including the last character of the shortest string, then the shortest string is considered the smaller.

Strings are equal only if their lengths as well as their contents are identical.

```
'A' = 'A' // is true  
'A' = 'a' // is false  
'2' <> ' 12' // is true  
'PASCAL' = 'PASCAL' // is true  
'PASCAL' = 'pascal' // is false  
'Pascal Compiler' <> 'Pascal compiler' // is true
```

The assignment operator is used to assign the value of a string expression to a string variable.

```
Age := 'twenty'  
Line := 'Many happy returns on your ' + Age + 'birthday'
```

If the maximum length of a string variable is exceeded (by assigning too many characters to the variable), the exceeding characters are truncated. E.g., if the variable *Age* above was declared to be of type **string[5]**, then after the assignment the variable will only contain the five leftmost characters: 'twent'.

## Concat restrictions

If the destination string is included in the concatenation the it **must** be the first string!

The following string concat result is not what you expect:

```
str:= str1 + str;
```

This is correct:

```
str:= str + str1;
```



## 4.1.10 ARRAY

An array is a structured type consisting of a fixed number of components which are all of the same type, called the *component type* or the *base type*. Each component can be explicitly accessed by indices into the array. Indices are expressions of any scalar type placed in square brackets suffixed to the *array identifier*, and their type is called the *index type*.

1..4 dimensions. One dimension can have up to 61440 (\$F000) members. Total size is limited to ca. 60kbytes. Types: bytes, int8, chars, booleans, words, integers, longwords, floats, pointers, procedures.

The definition of an array consists of the reserved word **array** followed by the index type, enclosed in square brackets, followed by the reserved word **of** followed by the component type.

```
type Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

```
var WorkHour : array[1..8] of Integer;  
    Week      : array[ 1 . 7] of Day;
```

```
type Players = (Player1, Player2, Player3, Player4);  
    Hand    = (One, Two, Pair, TwoPair, Three, Straight, Flush, FullHouse, Four, 5straightFlush);  
    Bid     = array[Players] of byte;
```

```
var Player   : array[Players] of Hand;  
    Pot      : Bid;
```

An array component is accessed by suffixing an index enclosed in square brackets to the array variable identifier:

```
Player[Player3] := FullHouse;  
Pot[Player3] := 100;  
Player[Player4] := Flush;  
Pot[Player4] := 50;
```

As assignment is allowed between any two variables of identical type, entire arrays can be copied with a single assignment statement.

The definition of an array constant consists of the constant identifier followed by a colon and the type identifier of a previously defined array type followed by an equal sign and the constant value expressed as a set of constants separated by commas and enclosed in parentheses.

```
type Status : array[0..2] of string[7] ;  
const Stat : Status = ('active', 'passive', 'waiting');
```

The example defines the array constant *Stat*, which may be used to convert values of the scalar type *Status* into their corresponding string representations.

The following boolean comparisons would return true:

```
Stat[0] = 'active'  
Stat[1] = 'passive'  
Stat[2] = 'waiting'
```



# AVRco Compiler-Manual

Multi-dimensional array constants are defined by enclosing the constants of each dimension in separate sets of parentheses, separated by commas. The innermost constants correspond to the rightmost dimensions.

```
type Cube = array[0..1,0..1,0..1] of integer;
```

```
const Maze : Cube = ( ( (0, 1) , (2, 3) ) , ( (4, 5) , (6, 7) ) );
```

```
type Tars = array[0..1, 3..4] of byte;
```

```
structconst {constant in Rom, at startup copied into Ram}  
ars : Tars = ( (0, 1) , ($FE, $FF) );
```

```
const {constant in Rom}  
arc : array[0..4] of word = (1, 200, 523, 1200, 9999);
```

```
var ar1 : array[5..12] of char;  
ar2 : array [6 .. 9, 56 .. 67] of word;  
ar3 : Tars;
```

```
x:= ar[1];  
ar1[8] := #7;  
ars [1, 3] := $7;  
ar2 [8, 60] := arc [3];
```

Array-manipulations only for Var and StructConst !

## Special construction

Array and Record constants can be read in from a file. The programmer is responsible of the contents of the file. The file length is not important. If the file is too short, the structure is filled with zeros. If the file is too long, the read-in is aborted at the Array/Record-limits.

```
Const Arr1 : array[0..31] of word = 'FileName.ext';
```

## 4.1.11 TABLE

1 dimension. Up to 255 members.

Types: bytes, Int8, chars, booleans, words, integers, longwords, floats, pointers, procedures.

TABLE is a specialized array which can contain look-up tables. The table length is limited on power of 2, to have a very fast access:

0..3, 0..7, 0..15, 0..31, .... 0..255.

The access automatically wraps, for example: an access to a table [0..7] with an index of 8 the access wraps to the table index 0. Tables have to be defined as global variables to assure a fast access. Table constants in ROM are not possible with AVR, but structured constants in ROM and EEPROM are possible. Accesses to a table must be done with **GetTable** and **SetTable**.

```
structconst {constant in Rom, at startup copied into Ram}  
Table1 : Table[0..3] = (0, $45, $A5, $FF);
```

```
var tb1 : Table[0..15] of char;  
tb2 : Table[0..127] of word;
```

```
x:= GetTable (tb1, 1);  
SetTable (tb1, x, $35);  
x:= GetTable (Table1, z);
```

## 4.1.12 RECORD

A record is a structure consisting of a fixed number of components, called *fields*. Fields may be of different type and each field is given a name the *field identifier*, which is used to select it.

### Record Definition

The definition of a record type consists of the reserved word `record` succeeded by a *field list* and terminated by the reserved word `end`. The field list is a sequence of *record sections* separated by semi-colons, each consisting of one or more identifiers separated by commas, followed by a colon and either a type *identifier* or a type *descriptor*. Each record section thus specifies the identifier and type of one or more fields

```
type tMonths = (Jan, Feb, Mar, Apr, May, Jun, July, Aug, Sep, Oct, Nov, Dec);
    Date = record
        Day    : byte;
        Month  : tMonths;
        Year   : word;
    end;
var
    Birth      : Date;
    WorkDay   : array[1..5] of date;
```

*Day*, *Month* and *Year* are field identifiers. A field identifier must be unique only within the record in which it is defined. A field is referenced by the variable identifier and the field identifier separated by a period.

```
Birth.Month := Jun;
Birth.Year := 1950;
WorkDay[Current] := WorkDay[Current-1];
```

Note that, similar to array types, assignment is allowed between entire records of identical types. As record components may be of any type, constructs like the following record of records of records are possible:

```
type Tmonths = (Jan, Feb, Mar, Apr, May, Jun, July, Aug, Sep, Oct, Nov, Dec);
    Tname = record
        FamilyName : string[32],
        ChristianNames : array[1..3] of string[16];
    end;

    TRate = record
        NormalRate,
        OverTime,
        NightTime,
        Weekend : Integer;
    end;

    TDate = record
        Day : byte;
        Month : TMonths;
        Year : word;
    end;

    TPerson = record
        ID : TName;
        Time : TDate;
    end;
```



# AVRco Compiler-Manual

```
Twages = record  
    Individual : TPerson;  
    Cost       : TRate;  
end;
```

```
Var Salary, Fee: TWages;
```

Assuming these definitions, the following assignments are legal:

```
Salary := Fee;  
Salary.Cost.Overtime := 950;  
Salary.Individual.Time := Fee.Individual.Time;  
Salary.Individual.ID.FamilyName := Smith;
```

The definition of a record constant consists of the constant identifier succeeded by a colon and the type identifier of a previously defined record type followed by an equal sign and the constant value expressed as a list of field constants separated by semi-colons and enclosed in parentheses. The constants must have the same name as in the Type-Declaration.

```
type Point = record  
    X, Y, Z : integer;  
end;
```

```
const APoint : Point = (X : 0; Y : 2; Z : 4);
```

The field constants must be specified in the same order as they appear in the definition of the record type.

## **Special construction**

Array and Record constants can be read in from a file. The programmer is responsible of the contents of the file. The file length is not important. If the file is too short, the structure is filled with zeros. If the file is too long, the read-in is aborted at the Array/Record-limits.

```
Const       Rec1 : TWages = 'FileName.ext';
```

### **4.1.12.1 WITH Statement for Access to Records**

The use of records as described above does sometimes result in rather lengthy statements; it would often be easier if we could access individual fields in a record as if they were simple variables. This is the function of the **with** statement: it 'opens up' a record so that field identifiers may be used as variable identifiers.

A **with** statement consists of the reserved word **with** followed by a record variable followed by the reserved word **do** and finally a statement. It ends up with the **EndWith** statement.

Within a with statement, a field is designated only by its field identifier, i.e. without the record variable identifier.

```
with Salary do  
    Individual := NewEmployee;  
    Cost := StandardRates;  
endwith;
```

## 4.1.13 PROCEDURE

16 bit (usually, depending on processor), parameter, word, address.  
“procedure“ declares a variable, which contains the address of a procedure.

```
var  proc : procedure;
```

```
procedure indirtest;  
begin
```

```
...  
end;
```

```
begin                {Main Program}
```

```
...  
Proc:= @indirtest;  {occupy variable with address of indirtest}  
Proc;                {call indirtest}
```

```
...  
end.
```

## 4.1.14 WORD

16 bit, 2 Bytes, 0..65535

Words are whole numbers. They are limited to a range of 0 through 65535.  
Words occupy two bytes in memory: low byte (lower address), high byte (higher address)

```
var  w      : word;  
      w1[24] : word;      {word at address 24}
```

## 4.1.15 INT8 or ShortInt

8bit, 1 Byte, -128..+127

Short integers (Int8) are whole numbers. They are limited to a range of - 128 through +127.  
Short integers occupy one byte in memory.

## 4.1.16 INTEGER

16bit, 2 Bytes, -32767..32768

Integers are whole numbers. They are limited to a range of - 32768 through 32767.  
Integers occupy two bytes in memory: low byte (lower address), high byte (higher address)

```
var  i      : integer;  
      w[@i] : word;      { word with the same address as i }
```

## 4.1.17 POINTER

16 bit (usually, depending on processor), 2 Bytes

Variables discussed up to now have been *static*, i.e. their form and size is pre-determined and they exist throughout the entire execution of the block in which they are declared. Programs, however, frequently need the use of a data structure which varies in form and size during execution. *Dynamic* variables serve this purpose as they are generated as the need arises and may be discarded after use.

Such dynamic variables are not declared in an explicit variable declaration like static variables, and they cannot be referenced directly by identifiers. Instead, a special variable containing the memory address of the variable is used to *point* to the variable. This special variable is called a *pointer variable*.

Sometimes it makes sense to use this type if you have not to access a variable by its name, but there is the possibility to work with the address of a variable.

An unqualified pointer always assigns to one byte, so it is “standardized”. An exception is the type conversion with `p := pointer(word1);`. This generated pointer is always untyped and is only qualified with an assignment to another pointer-variable. It is always better to work with qualified pointers. Therefore a private type is generated with

```
type tpb : pointer to Byte;
```

```
var pb : tpb;
```

```
pb := tpb (anyPointer);
```

A value to a pointer has to be assigned with address-operator @. A pointer is made invalid with the assignment `p := nil`. The pointer can itself be manipulated like every other variable. But be careful, a manipulated variable can also point to nirvana!!

A check of the validity of pointers (destination address) does not take place. The programmer should be very careful with pointers. If a pointer has possibly lost its validity, NIL should be assigned to it, to be able to test its validity when next used. But this must happen in the software, the compiler is not able to support this.

**Greetings from ‘C’!**

```
type tpw : pointer to word;
```

```
var p   : pointer to word; {always points to a word}  
    pb  : pointer;        {points to a byte}  
    pw  : tpw;           {points to a word}  
    b1  : byte;  
    b2  : byte;  
    w   : word;
```

```
Function DecWord (p : tpw) : word;
```

```
begin
```

```
  inc (p^);
```

```
  return(p^);
```

```
end;
```

**Procedure** *IncByte* (*b* : pointer to byte);

```
begin
  inc (b^);
  p:= pointer (b1);
end;
```

{Main}

```
...
pb:= @b1;
incByte (pb);
pb:= @b2;
incByte (pb);
p:= @w;
p^:= 1234;
p:= nil;
```

Sometimes it's necessary to declare a pointer before the object/type to which it points to, is declared. (e.g. linked lists). Then the pointer declaration must be defined with the attribute "Forward". The main declaration follows after the necessary type is defined:

```
type TPtr = pointer; forward; // preliminary declaration
      TRec1 = record
          ...
          Ptr1 : TPtr;
      end;
      TPtr = pointer to Trec1; // main declaration
```

Basically a pointer points into to address area of the CPU. With the AVR this is: Register-Page, IOPage, InternRam-Page and ExternRam-Page. All these areas reside in the linear address area from \$0000 to \$FFFF. As long as the programmer takes care of his pointer arithmetic, there are no problems (or rather: not many. Greetings from 'C' ). But you cannot directly access the Eeprom or Flash with a pointer. The Compiler cannot recognize that the result of a pointer manipulation now should point into the Flash. The Compiler must be informed that the access does **not** result in the normal address area. To this end the Compiler exports some predefined pointer types which can be used for a quasi type conversion:

```
EEpromPtr (pointer)
FlashPtr (pointer)
UsrDevPtr (pointer)
BankDevPtr (bank; pointer)
```

## **EEpromPtr**

An access by an *EEpromPtr*(pointer) is redirected into the EEprom with the address which the pointer contains. If the pointer points to \$100, the access is done to the address \$100 in the EEprom.

## **FlashPtr**

An access by an *FlashPtr*(pointer) is redirected into the ROM with the address which the pointer contains. If the pointer points to \$100, the access is done to the address \$100 in the Flash.

## **UsrDevPtr**

An access by an *UsrDevPtr*(pointer) is redirected into the UserDevice with the address which the pointer contains. If the pointer points to \$100, the access is done to the address \$100 in the UserDevice.

## BankDevPtr

An access by a *BankDevPtr*(*bnk*; *pointer*) is redirected into the Banked Device with the bank that *bnk* contains and the address which the pointer contains. If the pointer points to \$100, and *bnk* contains 2 the access is done to the address \$100 in bank 2 of the Banked Device.

```

type   TRec1   = record
                Rbb   : byte;
                Rww   : word;
                end;
        TPtr1   = pointer to TRec1;

const  FRec1   : TRec1 = (Rbb : $AA; Rww : $1234);

{$EEPROM}
var    ERec1   : TRec1;

{$UDATA}
var    URec1   : TRec1;

{$BDATA 2}
var    BRec1   : TRec1;

{$IDATA}
var    IRec1   : TRec1;
        Ptr1    : TPtr1;
        bb     : byte;
        ww     : word;
...
...

begin
    Ptr1:= @FRec1;
    bb:= FlashPtr (Ptr1)^.Rbb;
    ww:= FlashPtr (Ptr1)^.Rww;

    Ptr1:= @ERec1;
    EEpromPtr (Ptr1)^.Rbb:= $ff;

    Ptr1:= @URec1;
    UsrDevPtr (Ptr1)^.Rbb:= $ff;

    Ptr1:= @BRec1;
    BankDevPtr (2, Ptr1)^.Rbb:= $ff;

    Ptr1:= @IRec1;
    Ptr1^.Rbb:= $ff;
end.

```

### 4.1.17.1 Pointer AutoIncrement

One of the very few advantages of C over Pascal is the AutoIncrement of Pointers. This is also possible with AVRco by writing *Pointer^++*

The condition to use pointers with AutoIncrement is that they must be located either in the global space or local space (procedure/functions frame). You cannot manipulate pointers residing in the flash (obviously), in EEprom etc.

The pointer must be dereferenced. This means a read or write data move must be done with the pointer using the "^". An exception is *pointer++*;

Only typed pointers can be used with AutoIncrement.



The target/source of the pointer can be global, local, Flash or EEPROM variables and constants. But pay attention to the fact that the pointer always becomes incremented by the data size of the moved object. For a byte this is 1, with words this is 2, with arrays this is `sizeof(Array)` and with records this is also `sizeof(record)`. With strings the increment is not `length(string)` but `sizeof(string)`.

This all is the difference to `inc(pointer)` where the pointer is always incremented by 1. By the use of the AutoIncrement of pointers short and fast move loops can be build..

```
pointer^++:= variable;
pointer^++:= contant;
EEPromPtr(pointer)^++:= variable;
variable:= pointer^++;
variable:= FlashPtr(pointer)^++;
variable:= EEPROMPtr(pointer)^++;
pointer1^++:= pointer2^++;
etc.
```

## 4.1.18 LONGWORD

32bit, 4 Bytes, 0..4294967295

Longwords are whole numbers. They are limited to a range of 0 through 4294967295. Longwords occupy four bytes in memory.

```
var lw : longword;
```

## 4.1.19 WORD64 (\*4\*)(\*P\*)

64bit, 8 Bytes, 0..18446744073709551615

Word64 are whole numbers. They are limited to a range of 0 through 18446744073709551615. Word64 occupy eight bytes in memory.

```
var w64 : word64;
```

### **Attention:**

64 bit types like `word64`, `int64` and `double` are not available by default. The corresponding type has to be explicitly imported.

```
from System Import Word64, Double;
```

## 4.1.20 LONGINT

32 bit, 4 Bytes, -2147483648..2147483647

LongInts are whole numbers. They are limited to a range of -2147483648 through 2147483647. LongInts occupy four bytes in memory.

```
var li : longint;
```



## 4.1.21 INT64 (\*4\*)(\*P\*)

64bit, 8 Bytes, -9223372036854775808 ... 9223372036854775807

Int64 are whole numbers. They are limited to a range of -9223372036854775808 through 9223372036854775807. Int64 occupy eight bytes in memory.

```
var i64 : int64;
```

### **Attention:**

64 bit types like word64, int64 and double are not available by default. The corresponding type has to be explicitly imported.

```
from System Import Int64, Double;
```

## 4.1.22 FLOAT

32 bit, 4 Bytes, 6..9 Digits, 10E-38..10E38

```
var f : float;
```

### **Attention:**

32 bit types like longint, longword and float are not available by default. The corresponding type has to be explicitly imported.

```
from System Import LongInt, Float;
```

## 4.1.23 DOUBLE (\*4\*)(\*P\*)

64bit, 8 Bytes, 10E-308..10E308

```
var d : double;
```

### **Attention:**

64 bit types like word64, int64 and double are not available by default. The corresponding type has to be explicitly imported.

```
from System Import Int64, Double;
```

## 4.1.24 ENUM

8bit, 0..255 Enumeration

```
type eKey = (Key1, Key2, Key3); {type declaration}  
var Keys : eKey; {variable of type eKey}  
 {runs from Key1..Key3}
```

```
if Key2 in Keys then ...
```

It's possible to change enumerations with the INC and DEC procedures. The upper and lower limits are taken in account, but unlike SUCC and PRED these functions wrap around at the limits.

Enumeration can also be defined with holes or gaps:

**type**

```
tEnum = (aaa, bbb, ccc=100, ddd, eee=200, fff);  
0 1 100 101 200 201 <- numerical values
```

## 4.1.25 SEMAPHORE

8bit, Byte is imported by processes and tasks

**var** sema : Semaphore;

Semaphores are specialized variables and serve for the process-synchronization.

A process or a task can increment or decrement a semaphore. A value > 0 may mean for a task/process for example that a certain function/procedure is disabled.

An access can only be done with the corresponding special functions/procedures, because the access has to be secured from interrupts (task changes).

**State of a Semaphore (byte)**

**Function** SemaStat (sema : semaphore) : byte;

**Incrementing of a Semaphore**

**Procedure** IncSema (sema : semaphore);

**Decrementing of a Semaphore**

**Function** DecSema (sema : semaphore) : boolean;

**Process is waiting for Semaphore > 0**

**Function** WaitSema (sema : semaphore [; timeout: word]) : boolean;

The TimeOut parameter is optional. If omitted the process waits until the event occurs. The same is true if TimeOut is set to 0000. With a value > 0 the Wait is terminated after (TimeOut \* SysTicks).

The function result becomes true if there was no Timeout.

A TimeOut in Tasks is not possible. This parameter is ignored.

## 4.1.26 PIPE

generic is mainly used by processes or tasks

### 4.1.26.1 Pipe for ordinal Types

A pipe is a block of memory, which is organized as a so called FIFO (First in, First Out) or queue. The pipe can be a container for all ordinal types (boolean..longint) and float. The length of the pipe or the possible number of parameters/variables is max. 255. An access to pipes can only be done with the corresp. functions/procedures, because the access has to be secured from interrupts (task changes).

**Declaration:**

**var** Pipe1 : Pipe[16] of byte; // int8, boolean, word, integer etc.

**State of a Pipe (byte)** = number of available parameters

**Function** PipeStat (pipe1 : pipe) : byte;

The function PipeStat is also applicable to RxBuffer, RxBuffer1, -2, -3 and TxBuffer, TxBuffer1, -2, -3 of the serial interfaces.

**State of a Pipe (boolean)**

**Function** PipeFull (pipe1 : pipe) : boolean;



# AVRco Compiler-Manual

## Add Parameter to Pipe

**Function** *PipeSend* (*pipe1* : pipe; *parm* : type) : boolean;

## Fetch Parameter from Pipe

**Function** *PipeRecv* (*pipe1* : pipe) : type;

Waits until the pipe at least contains one parameter and then returns the oldest entry.

**Function** *PipeRecv\_ND* (*pipe1* : pipe) : type;

This function allows the read out of a pipe without removing or changing the pipe contents "non-destructive-readout".

## Empty Pipe

**Procedure** *PipeFlush* (*pipe1* : pipe);

## Process or Task waiting for Pipe

**Function** *WaitPipe* (*pipe1* : pipe [, *timeout*: word]) : boolean;

The TimeOut parameter is optional. If omitted the process waits until the event occurs. The same is true if TimeOut is set to 0000. With a value > 0 the Wait is terminated after (TimeOut \* SysTicks). The function result becomes true if there was no Timeout.

### 4.1.26.2 Pipe of Bit

To save the limited resources and memory the Pipe type is able to handle bits:

*BitPipe* : pipe[xx] of bit;

The behavior of this Pipe is the same as with a Pipe of boolean. But only one bit is used for each entry. Because of this each 8 Bits result in a memory usage of one byte. The access follows the FIFO principle. The first bit written to the pipe is also the first bit read from the pipe. A pipe can handle upto 248 bits which results in a maximal memory consumption of 36 Bytes.

### 4.1.26.3 Pipe for complex Types

It's possible to build pipes of strings, arrays and records.

To read these complex types the function *PipeRecv* is expanded:

**Function** *PipeRecv* (*Pp* : Pipe; **var** *Value* : PipeType{record, array, string} [, *doWait* : boolean]) : boolean;

The parameter Record, Array or String must be an already defined type.

The name of the target variable must be passed as the second parameter. With the optional switch "doWait" it's possible to define whether the function returns with a FALSE, if nothing there, or waits in a loop until the pipe has any content. If the switch "doWait" is omitted the function acts like the "doWait" is defined as true and the result is always a true.

**Function** *PipeRecv\_ND* (*P* : Pipe; **var** *Value* : PipeType{record, array, string} [, *doWait* : boolean]) : boolean;

The same as the function above except that the read out of a pipe doesn't remove or change the pipe contents. "non-destructive-readout".

### 4.1.27 SYSTIMER

16 bit, word is imported by SysTick

Variable is decremented by every system tick, if it is > 0.

**var** *Timer1* : SysTimer[, UpCount]; {variable of type SysTimer}

*SetSysTimer* (*Timer1*, 50000);

**repeat until** *GetSysTimer* (*Timer1*) = 0;

( better is -> )

```
repeat until isSysTimerZero (Timer1);
```

For simple time measure functions an upcounter is very useful. This timer must be started with *ResetSysTimer*. The system now increments this timer with each SysTick. An overflow/wrap is inhibited. The max value is limited to \$FFFF.

The SysTimer functions "*GetSysTimer, ResetSysTimer, SetSysTimer*" are also applicable to the upcounters, but the function "*isSysTimerZero*" obviously is not.

The system supports a total count of max. 16 SysTimer (SysTimer +SysTimer8+SysTimer32).

## 4.1.28 SYSTIMER8

8 bit, byte is imported by SysTick

Variable is decremented by every system tick, if it is > 0.

```
var Timer2 : SysTimer8[, UpCount]; {variable of type SysTimer8}
```

```
SetSysTimer (Timer2, 50);
```

```
repeat until isSysTimerZero (Timer2);
```

Interrupts are not disabled during an access. So an access to this Timer type is faster and shorter as the access to a "SysTimer".

For simple time measure functions an upcounter is very useful. This timer must be started with *ResetSysTimer*. The system now increments this timer with each SysTick. An overflow/wrap is inhibited. The max value is limited to \$FF.

The SysTimer functions "*GetSysTimer, ResetSysTimer, SetSysTimer*" are also applicable to the upcounters, but the function "*isSysTimerZero*" obviously is not.

The system supports a total count of max. 16 SysTimer (SysTimer +SysTimer8+SysTimer32).

## 4.1.29 SYSTIMER32

32 bit, 4 bytes is imported by SysTick

Variable is decremented by every system tick, if it is > 0.

So extreme long times/delays can be achieved. If used, LongWords must be imported.

```
from System import longword;
```

```
var Timer3 : SysTimer32[, UpCount]; {variable of type SysTimer32}
```

```
SetSysTimer (Timer3, 100000);
```

```
repeat until isSysTimerZero (Timer3);
```

For simple time measure functions an upcounter is very useful. This timer must be started with *ResetSysTimer*. The system now increments this timer with each SysTick. An overflow/wrap is inhibited. The max value is limited to \$FFFFFFFF.

The SysTimer functions "*GetSysTimer, ResetSysTimer, SetSysTimer*" are also applicable to the upcounters, but obviously the function "*isSysTimerZero*" is not.

The system supports a total count of max. 16 SysTimer (SysTimer +SysTimer8+SysTimer32).

## 4.1.30 PIDCONTROL

Pseudo-Record

PID-controller, is often used in technical applications, for example temperature controlling, servos, rotation speed controller, etc.

PID-controller have two input parameters: *nominal value* = the required value and the *actual value*. Four parameters, which are usually only adjusted once, are *pFactor*, *iFactor*, *dFactor* and *sFactor*. The controller output, which belongs to the actuator (heating, motor etc), is calculated by the function 'execute'.

The controller type is determined by the two initializing parameters '*iLimit*' and '*dIntVal*'. The internal variables *pValue*, *iValue*, *dValue* can be read but should not be changed.

### **iLimit**

is of type longword (0..100000) and determines the maximal size of the **I**-clipping. If *iLimit* = 0 the **Integral**-value will not be calculated and will be discarded (e.g. PD-controller).

### **dIntVal**

is of type byte (0, 1, 2, 4, 8, 16, 32) and determines the degree step of calculation of the **D**-clipping (gradation).

If *dIntVal* = 0 the **Differential**-value of the controller will not be calculated and will be discarded (e.g. PI-controller).

If the value = 1, the gradation will be calculated from the last error value up to the actual error value.

In the remaining cases a corresponding array is introduced. So it is possible to calculate the gradation with just a few values required.

The controller itself calculates with **longinteger**. In practice it is improbable to get an overflow with execute.

```
var Pid1 : PIDcontrol[iLimit, dIntVal];
```

```
{Init}
```

```
Pid1.pFactor:= 1000;
```

```
Pid1.iFactor:= 2500;
```

```
Pid1.dFactor:= 678;
```

```
Pid1.sFactor:= 10000;
```

```
{Run}
```

```
Pid1.Actual:= 500;
```

```
Pid1.Nominal:= 550;
```

```
PWM1:= Pid1.Execute;
```

## 4.2 Operators

### 4.2.1 NOT

*a := not a; {inverting bits of var a }*

Only the types byte, Int8, boolean, integer, word, longint and longword are permitted as operands. If the variable 'a' has the value \$FF in the example above, then it has the value \$00 after the operation. See also the **Negate** function.

### 4.2.2 DIV

*a := a div b; {integer division}*

Only the types byte, Int8, integer, word, longint and longword are permitted as operands. If the variable 'a' has the value \$10 and 'b' the value \$02 in the example above, then 'a' has the value \$08 after the operation.

### 4.2.3 MOD

*a := a mod 5; {Modulo of integers}*

Only the types Byte, Int8, Integer, word, Longint and Longword are permitted as operands. If the variable 'a' has the value \$06 in the example above, 'a' has the value \$01 after the operation. Modulo is the remainder after integer division.

### 4.2.4 AND

*a := a and \$0f; {And Mask}*

Only the types Byte, Int8, Integer, word, Longint and Longwords are permitted as operands. If the variable 'a' has the value \$13 in the example above, then 'a' has the value \$03 after the operation. Thus the result of and is those bits that are set in the operand and in the mask. 'And' is also used with boolean functions.

*if (a > b) and (a < c) then ... endif;*

### 4.2.5 OR

*a := a or \$30; {Or Mask}*

Only the types Byte, Int8, Integer, word, Longint and Longwords are permitted as operands. If the variable 'a' has the value \$09, then 'a' has the value \$39 after the operation. 'Or' is additional setting the bits in the result, which were set in the mask.

'Or' is also used with boolean functions.

*if (a > b) or (a < c) then ... endif;*

### 4.2.6 XOR

*a := a xor 1; {Xor Mask}*

Only the Types Byte, Int8, Integer, word, Longint and Longword are permitted as operands. If the variable has the value \$01, then 'a' has the value \$00 after the operation. 'Xor' is exclusive or, setting bits that are the same in the mask and in the operand to zero, and those that are different to one.

## 4.2.7 SHL

*a := a shl 5; {shift left}*

Only the types Byte, Int8, Integer, word, Longint and Longword are permitted as operands. If the variable has the value \$03, then 'a' has the value \$60 after the operation. 'SHL' shifts all Bits in the operand left and filling the bits from right with 0.

## 4.2.8 SHLA

*a := a shla 5; {shift left arithmetically}*

Only the Types Byte, Int8, Integer, word, Longint and Longword are permitted as operands. If the variable 'a' has the value \$03 in the example above, then 'a' has the value \$60 after the operation. 'SHLA' shifts all bits within the operand to left and fills the bits from right with '0'. In contrast to 'SHL' the highest value bit (sign bit) stays unchanged. So it is ensured that a negative number stays negative.

## 4.2.9 SHR

*a := a shr 4; {shift right}*

Only the types Byte, Int8, Integer, word, Longint and Longword are permitted as operands. If the Variable 'a' has the value \$81, then 'a' has the value \$08 after the operation. 'SHR' shifts all Bits in the operand right and fills up the bits from left with 0.

## 4.2.10 SHRA

*a := a shra 4; { shift right arithmetically}*

Only the Types Byte, Int8, Integer, word, Longint and Longword are permitted as operands. If the variable 'a' has the value \$71 in the example above, then 'a' has the value \$07 after the operation. 'SHRA' shifts all bits within the operand right and fills the bits from left with '0'. In contrast to 'SHR' the highest value bit stays unchanged. Thus it is ensured, that a negative number stays negative.

## 4.2.11 ROL

*a := a rol 4; {rotate left}*

Only the Types Byte, Int8, Integer, word, Longint and Longword are permitted as operands. If the Variable 'a' has the value \$81, then 'a' has the value \$18 after the operation. 'ROL' rotates all bits to the left. All bits are retained, but change their positions

## 4.2.12 ROR

*a := a ror x; {rotate right}*

Only the types Byte, Int8, Integer, word, Longint and Longword are permitted as operands. If the Variable 'a' has the value \$01, so 'a' has the value \$02 after the operation. 'ROR' rotates all bits to the right. All bits are obtained, but change their positions.



## 4.2.13 IN

*if v2 in ['a'..'g'] then ...*                    *{analyze Enum}*  
*if Key in [Key2..Key4] then ...*            *{analyze Enum}*  
*if x in[45..56] then ..*

All ordinal types and enums are permitted as arguments Byte, Char, ... LongInt, as well as float.

## 4.2.14 +

*a:= a + 5;*                    *{add}*

As operands only the types Byte, Int8, Integer, word, Longint, Longword and Float are permitted. A special case is a string-operation.

## 4.2.15 -

*a:= a - b;*                    *{subtract}*

As operands only the types Byte, Int8, Integer, word, Longint, Longword and Float are permitted. If the minus should be used as sign, this **has to be** enclosed by a bracket.

*a:= a \* (- b);*                *{multiply with negative value}*

## 4.2.16 /

*a:= a / 5.5;*                *{Float Division}*

## 4.2.17 \*

*a:= a \* %11000100;* *{Multiplication}*

*f:= f \* 1.5;*

## 4.3 Pseudo Operators

### 4.3.1 @

*p := @a;                    {memory location address}*

Only variables, procedures and functions are permitted as operands, because only they have a physical address. After the operation 'p' contains the address of 'a'. Actually '@' is not an operator, but a system function. The destination normally is a pointer.

### 4.3.2 ^

*x := p^;                    {variable from pointer}*

Only variables of the type pointer are permitted as operands. After the operation 'x' contains the value, which is in the memory location 'p' that points to. Actually '^' is not an operator, but a system function.

Further a prefixed '^' in front of a char is the instruction for the compiler to interpret the following symbol as an ANSI control character.

A **^G**, for example, produces the so called bell-symbol (hex 09). The compiler executes the following operation: *result = 'G' - '@'*. (\$49 - \$40).

*const Bell = ^G;        {Control G}*

### 4.3.3 #

*const LF = #10;        {Line Feed}*

The number-symbol is another way (in addition to ^), to define control symbols (non representable letters). The argument, which follows the symbol, must be a decimal number on the area of 0..255. It defines the following number as a character.

'#' is actually not an operator, but a system function.

### 4.3.4 \$

*const x1 = \$10;        {decimal 16}*

The dollar symbol defines the following constant to a hexadecimal value.

### 4.3.5 %

*const b1 = %10100101;    {hex \$A5, decimal 165}*

Percent symbol defines the following constant to a binary value.

## 4.4 User Defined Language Elements

### 4.4.1 Identifier

Identifiers are used to denote labels, constants, types, variables, procedures, and functions. An identifier consists of a letter followed by any combination of letters, digits, or underscores. An identifier is limited in length to 64 characters, and all characters are significant.

```
PASCAL
square
persons_counted
BirthDate
3rdRoot           // illegal, starts with a digit
Two Words         // illegal, must not contain a space
```

As the AVRco does not distinguish between upper and lower case letters, the use of mixed upper and lower case as in *birthDate* has no functional meaning. It is nevertheless encouraged as it leads to more legible identifiers. *VeryLongIdentifier* is easier to read for the human reader than *VERYLONGIDENTIFIER*

### 4.4.2 Numbers

Numbers are constants of byte, Int8, integer type or of float type. Integer constants are whole numbers expressed in either decimal, hexadecimal or binary notation. Hexadecimal constants are identified by being preceded by a dollar sign: \$1234 is a hexadecimal constant. Binary constants are preceded with a percentage sign: %1011100 is a binary constant.

The decimal longint range is -2147483648 to +2147483647, the hexadecimal integer (longint) range is \$0 to \$FFFFFFFF. The binary integer (longint) range is %0 to %11111111111111111111111111111111 (32 \* 1).

```
12345
-1
$123
$ABC
$123G           // illegal, G is not a legal hexadecimal digit
%1011
%1003           // illegal, 3 is not a legal binary digit
1.2345         // illegal as integer, contains a decimal part
```

### 4.4.3 Strings

A string constant is a sequence of characters enclosed in single quotes:

```
'This is a string constant'
```

Strings containing only a single character may be of the standard type *char*. The actual type is determined by the context.

```
'PASCAL'
'You"ll see'
''
```

As shown in the examples, a single quote within a string is written as two consecutive quotes. The quotes enclosing no characters, denoting *the empty string* - is compatible only with string types (**not** with the type *char*).



# AVRco Compiler-Manual

## 4.4.4 Control Characters

The AVRco also allows control characters to be embedded in strings.

The # symbol followed by an byte constant in the range 0..255 denotes a character of the corresponding ASCII value.

The AVRco allow the use of the ^ character followed by an alpha character to denote control characters.

```
#10      // ASCII 10 decimal (Line Feed).
#$1B    // ASCII 1B hex (Escape).
^G      // ASCII 07 hex (Bel)
```

Sequences of control characters may be concatenated into strings by writing them using a + between the individual characters:

```
#13 + #10
#27 + #20
```

Control characters may also be mixed with text strings:

```
'Waiting for input!'+#7+#7+#7+#7+'Please wake up'
```

## 4.4.5 Comments

A comment may be inserted anywhere in the program where a delimiter is legal. It is delimited by the curly braces { and }, which may be replaced by the symbols (\* and \*).

It is also possible to use the symbols // to comment the remainder of the line as can be done in the “C” language.

```
{This is a comment}
(* also a comment *)
```

Curly braces may be nested within curly braces, and (\* \*) may be nested within (\*. \*).

Curly braces may be nested within (\* \*) and vice versa, thus allowing entire sections of source code to be commented away, even if they contain comments.

## 4.5 Expressions

Expressions are algorithmic constructs specifying rules for the computation of values. They consist of operands, variables, constants, and function designators combined by means of operators as defined in the following.

### 4.5.1 Operators

- 1) Operators fall into five categories, denoted by their order of precedence:
- 2) Unary minus (minus with one operand only)
- 3) Not operator.
- 4) Multiplying operators: \*, /, div, mod, shl, shr.
- 5) Adding operators: +, -, or, and, xor.
- 6) Relational operators: =, < > , > , < , <= , >= , in.

Sequences of operators of the same precedence are evaluated from left to right. Expressions within parentheses are evaluated first and independently of preceding or succeeding operators. The operators must be the same type. There is no automatic type casting.

#### **Attention**

The order of the operators (\* AND SHR etc.) is not completely implemented yet. So please use always parentheses in conjunction with expressions.

#### 4.5.1.1 Unary Minus

The unary minus denotes a negation of its operand which may be of *Float*, *Longint*, *Int8* or *Integer* types.

#### 4.5.1.2 Not Operator

The not operator negates (inverses) the logical value of its Boolean operand:

*not True = False*

*not False = True*

In AVRco Pascal the “not” operator can be used to invert a value if the argument is of type byte, integer, longint, word or longword.

#### 4.5.1.3 Multiplying Operators

<u>Operator</u>	<u>Operation</u>	<u>Types</u>
*	multiplication	Float, Longint, Longword, Word, Integer, Int8, Byte, Bitset
/	division	Float, Bitset
div	division	Longint, Longword, Word, Integer, Int8, Byte
mod	modulus	Longint, Longword, Word, Integer, Int8, Byte
and	arithmetic and	Longint, Longword, Word, Integer, Int8, Byte
and	logical and	Boolean
shl	shift left	Longint, Longword, Word, Integer, Int8, Byte
shr	shift right	Longint, Longword, Word, Integer, Int8, Byte
rol	rotate left	Longint, Longword, Word, Integer, Int8, Byte
ror	rotate right	Longint, Longword, Word, Integer, Int8, Byte

```

12 * 34           // = 408
123 / 4           // = 30.75
123 div 4         // = 30
12 mod 5          // = 2
True and False   // = False
12 and 22         // = 4
2 shl 7           // = 256
256 shr 7         // = 2
    
```

## 4.5.1.4 Adding Operators

Operator	Operation	Types
+	addition	Float, Longint, Longword, Word, Integer, Int8, Byte, String, Char, Bitset
-	subtraction	Float, Longint, Longword, Word, Integer, Int8, Byte, Bitset
or	arithm. or	Longint, Longword, Word, Integer, Int8, Byte, Bitset
or	logical or	Boolean
xor	arithm. xor	Longint, Longword, Word, Integer, Int8, Byte
xor	logical xor	Boolean

```

123+456      // = 579
456-123      // = 333
True or False // = True
12 or 22     // = 30
True xor False // = True
12 xor 22    // = 26

```

## 4.5.1.5 Relational Operators

The relational operators work on all standard scalar types: *Float, Longword, Longint, Word, Integer, Boolean, Char, Int8, and Byte*. The type of the result is always Boolean, i.e. *True* or *False*.

```

=      equal to
<>    not equal to
>      greater than
<      less than
>=    greater than or equal to
<=    less than or equal to

```

```

a = b      // true if a is equal to b
a <> b     // true if a is not equal to b
a > b      // true if a is greater than b
a < b      // true if a is less than b
a >= b     // true if a is greater than or equal to b
a <= b     // true if a is less than or equal to b

```

A comparison of record and array types is also possible.

In this case the variables must consist of the same type. This means they must be predefined as types:

```

Type TRec = record
    ...
end;

```

```

var Rec1, Rec2 : TRec;

```

```

if Rec1 <> Rec2 then ...
bool := Rec1 = Rec2;

```

## 4.5.2 Function Designators

A function designator is a function identifier optionally followed by a parameter list, which is one or more variables or expressions separated by commas and enclosed in parentheses. The occurrence of a function designator causes the function with that name to be activated. If the function is not one of the pre-defined standard functions, it must be declared before activation.

```

Z:= func1(x); // assume func1 is a declared function returning a value
F:= sqr(a);   // sqr is a builtin function returning the square of a value

```

## 4.6 Keywords

### 4.6.1 PROGRAM

Starting the program. A particular form of program framework is necessary. This starts with 'Program *name*' and ends with 'End.'

<b>Program</b> Test;	
<b>Device</b> ...	{hardware declaration}
<b>Import</b> ...	{system Functions}
<b>from</b> system <b>Import</b> ...	{types and functions}
<b>Define</b> ...	{hardware definition}
<b>Implementation</b>	{program start}
<b>const</b> ...	{constant declaration}
<b>var</b> .. .	{variable declaration}
<b>Procedure</b> System_Init;	{private initialize}
<b>begin</b>	
...	
<b>end;</b>	
<b>Interrupt</b> Timer1;	{interrupt declaration}
<b>begin</b>	
...	
<b>end;</b>	
<b>Process</b> Pxx (14,20 : iData);	{process declaration}
<b>begin</b>	
...	
<b>end;</b>	
<b>Task</b> Txx (iData);	{task declaration}
<b>begin</b>	
...	
<b>end;</b>	
<b>Procedure</b> ABC (z : integer);	{procedure head}
<b>Var</b> xy : byte;	
<b>begin</b>	
...	
<b>end;</b>	
<b>Function</b> CDE : boolean;	{function head}
<b>Var</b> a, b : byte;	
<b>begin</b>	
...	
<b>Return</b> (a > b);	{result of the function}
<b>end;</b>	
<b>begin</b>	{main = main program}
ABC (i);	
x:= CDE;	
<b>end.</b>	

## **Comment:**

The **standard version** of the compiler has no linker. So it is not possible, to introduce units or to link external code. At first sight this looks like a limitation. But in practice this is not really a disadvantage, because the generated code and so the program size is limited by the relative small program memory (typ. 8kB) of the destined CPU.

Frequently recurring program parts can be included with help of the compiler switch “Include“ `{$/ ... }` and `{$.}`.

Further since the often small program memory does not allow extensive stack- or frame-operations, this limits the number of local variables in procedures and functions.

In contrast to well-known programmer-systems for the PC (Turbo Pascal, Delphi etc.), in which the compiler knows exactly the hardware, a compiler for embedded systems (SingleChips) does not know the target hardware unless it is told.

Hence it is necessary for the programmer to specify the destined hardware/CPU exactly. Because of the often limited system resources (RAM/ROM) it must further be considered which functions of the compiler (system) are really required.

To specify the available hardware and the required functions for the compiler, you need to have diverse instructions (Device, Import and Define). These three definitions are required and **have to be specified** in above mentioned order.

To identify the exact position of RAM and ROM and their sizes as well as the implemented hardware of the CPU (SCI, ADC etc) the compiler needs a **Description-File**, which contains this information. These are provided by E-LAB. An example be found in *P90S851 5.dsc*, for instance.

## **4.6.2 DEVICE**

Processor and Hardware Specification.

The arguments of “Device“ are totally processor specific. The processor has to be the first entry. Depending on the processor family the parameters of the compiler are evaluated in the assembler or in both program parts. In part the data is included in HexFile and is required by the programming unit. The Device parameter names correspond to the respective processor databooks where they are also specified.

The used CPU-name has to be identical with the processor-control file prefixed by a ‘P’.

Example for a Device instruction of the AVR AT90S2313:

```
device = 90S2313;
```

The relevant control file must be named: P90S2313.dsc



## 4.6.3 IMPORT

Import of **Hardware dependant** system functions

As mentioned above, the sometimes modest resources of some implemented processors force us to include only the most necessary system functions. Memory or program intensive and also more exotic functions have to be imported explicitly.

Certain imports again require an additional specification, which is made by means of the *Define* section. Some functions import additional functions automatically, which possibly have to be further specified within the define section. The description of the separate import functions can be found in the **Driver Manuals** of the respective drivers.

Many functions assume corresponding Hardware within the CPU. SysTick requires a Hardware Timer and SwitchPort a port. Further some imports require the SysTick, e.g. ADCport. So SysTick should always be imported first

*Import* SYSTICK, SWITCHPORT1;

## 4.6.4 FROM

With FROM a type, function or procedure is imported from a specific source.

*From System Import* LongInt, LongWord, Float;

Possible imports of **types** and system functions:

**Double**  
**Float**  
**Int64**  
**LongInt**  
**Word64**  
**LongWord**  
**Processes** Sleep, Suspend, Resume, Priority, WaitSema ...  
**Tasks** Sleep, Suspend, Resume, Priority, WaitSema ...  
**Pipes** PipeFlush, PipeSend, PipeRecv, PipeStat, PipeFull ...  
**Pids** pFactor, iFactor, dFactor, sFactor, Actual, Nominal, Execute

*From System Import* LongInt, LongWord, Float, Pipes;

## 4.6.5 DEFINE

Parameters for certain import functions

Certain imports require an additional specification, which is specified using *Define*. Here for example a time in msec is assigned to the system tick or a physical port address is assigned to the switch port. At the very least the processor speed **ProcClock** has to be specified.

The description of the defines can be found in the **Driver Manuals** of the respective drivers.

*Define* ProcClock = 4000000; {4 Mhz}  
 SYSTICK = 10; {10msec}  
 StackSize = 82, iData; {82 Bytes in iData}  
 FrameSize = 99, xData; {99 Bytes in xData}  
 SwitchPort1 = PortA; {Port Addr}  
 SerPort = 9600, Stop2; {9600Bd, 2 Stopbits}  
 RxBuffer = 8, iData; {RxBuffer 8 Chars}  
 TxBuffer = 10, iData; {TxBuffer 10 Chars}  
 PWMpresc1 = 4; {prescaler 4}



# AVRco Compiler-Manual

## Tip:

Most of the Defines, which consist of a numeric constant, can be used as a normal constant in the application.

```
Define ProcClock = 4000000;           {4 Mhz }  
        SYSTICK = 10;                 {every 10msec }  
        ...
```

```
const myConst = ProcClock div SysTick;
```

```
if ProcClock > 4000000 then ...
```

It's also possible to make a "Forward" reference at the point where the Define is searched by the compiler. This forward can be completely defined later in the implementation part of the program or unit. An example how to do this can be found in the directory `..\E-Lab\AVRco\Demos\Mega161`.

## 4.6.6 Hardware Imports within Units

If Units are imported the definitions can also be placed into an Unit:

### Main Program

```
Import SysTick, MatrixPort, SerPort;
```

```
From System Import longword, longint, float, pipes;
```

### **Define**

```
ProcClock = 8000000; {Hertz}  
SysTick   = 10;      {msec}  
StackSize = $0020, iData;  
FrameSize = $0040, iData;  
SerPort   = 9600;  
RxBuffer  = 16, iData;
```

```
DefineFrom unit1; // Unit1 defines the Matrixport
```

### Unit

```
Unit Unit1;
```

### **Define**

```
MatrixRow = PortD, 4; {use PortD, start with bit4}  
MatrixCol = PinD, 0; {use PinD, start with bit0}  
MatrixType = 3, 4;   {3 Rows at PortD, 4 Columns at PinD}
```

### **Interface**

...

The reserved word "*DefineFrom*" within the main "Define" block switches the scanning from main program to the given Unit name, where the scanning of the defines continues. When the word "*Interface*" appears the scanning is switched back to the main program.

## 4.6.7 DEFINE\_USR

With "Define\_USR" it is possible to define constants which are visible and accessible from every point of the program and also from each Unit. But there can be only ordinal types.

This Define should only be used if it is absolute necessary because it's not a good programming style. The better way is to place such globals into a Unit which resides in the last position of the Unit chain. Then the definitions are also visible from all other parts of the application.

```
Define SysTick = 10;
```

```
...
Define_USR myConst = 1;
           myBool= true;
```

## 4.6.8 DEFINE\_FUSES

As an option the fuse settings for the programmers can be defined in the source. The definition must be placed between the "Device" and the "Import" statement.

The fuse names correspond with the names in the datasheet of the CPU and the names found in the programmer software. Spaces in the origin names must be replaced by "\_" .

The four possible fuse groups are named  
"LockBits0, FuseBits0, FuseBits1, FuseBits2"

All fuses are low-active, this means if a fuse name is given, e.g. "CKSEL1", so this bit will be programmed to **ZERO** (low = active).

So fuses which do not appear are always programmed to "1" (high = inactive)!

For example the statement:

```
LockBits0 = [];  
programs all Lockbits to "1" = inactive.
```

The statement:

```
FuseBits0 = [CKSEL1];  
programs the fusebit CKSEL1 to "0", all other bits of this group get the value "1" = inactive.
```

With creating a new project these fuses are written into the ISPE-file for the programmer's purposes.

If the ISPE file already exists, it will be not changed. The optional define of "OverRide\_Fuses" always forces an ISPE update. *OverRide\_Fuses* must be placed after *Define\_Fuses*.

The programmer software copies the settings in the ISPE-file to the "write boxes" of the fuses. The general options

**program Fuses** and  
**program Lockbits**

in *Programmer Options* are not changed by that, except bthey are explicetely listed here. The user selects here whether the fuses are witten which each programming cycle.

The *NoteBook* entry is used only with the professional stand alone programmers.

With an extreme large project count the selection of the desired project becomes for these programmers somewhat difficult. Because of this a tabbed notebook is implemented. It contains the pages A...N. The above Define now selects the notebook page where the application must be stored.

*ProgMode* defines one of the 3 possible programming modes.

*Supply* instructs the programmer to provide a voltage with a current limit to the target device.



# AVRco Compiler-Manual

*AddApp* instructs the programmer to load additional hexfiles build by another project:

```
AddApp = 'pathname\projectname';
```

This provides joining a BootApplikation with the Main application in the programmer and both are programmed together in one operation. See also Demos\BootApplication

```
Device = mega16, VCC = 5;
```

## **Define\_Fuses**

```
Override_Fuses; // optional, always replaces fuses in ISPE  
COMport = COM1; // COM2..COM7, USB  
Supply = 4.0, 200; // programmer supplies target, 4.0Volt, 200mA  
LockBits0 = [];  
FuseBits0 = [CKSEL1];  
FuseBits1 = [BOOTRST, BOOTSZ1, SPIEN, OCDEN];  
ProgMode = SPI; // SPI, JTAG or OWD  
ProgFuses = true; // or false – program Fuse Bits (* REV4 *)  
ProgLock = true; // or false – program Lock Bits (* REV4 *)  
ProgFlash = true; // or false – program Flash (* REV4 *)  
ProgEeprom = true; // or false – program Eeprom (* REV4 *)  
AddApp = 'pathname\projectname';
```

```
Import SysTick, ...;
```

```
Define SysTick ...
```

## 4.6.9 IMPLEMENTATION

Starting the program.

The compiler inserts the reset-code and the initialization at this point. The instruction must exist and has to be placed directly after the define-bloc.

Normally the global type-, constant-, and var declarations follow the implementation instruction.

## 4.6.10 TYPE

Starting of a type declaration

A data type in Pascal may be either directly described in the variable declaration part or referenced by a type identifier. Several standard type identifiers are provided like *boolean*, *byte*, *word*, *integer* etc., and the programmer may create his own types through the use of the type definition.

The reserved word **type** heads the type definition part, and it is followed by one or more type assignments separated by semicolons. Each type assignment consists of a type identifier followed by an equal sign and a type.

The definition of ones own types is a good programming practice and makes for better readability.

```
type tpb    = pointer to byte;
      tarr   = array[2..7] of byte;
      tpa    = pointer to tarr;
      tstr   = string[8];
      tps    = pointer to tstr;
      tKey   = (Forw, Stop, BackW);
```

```
var pb     : tpb;
      pa     : tpa;
      ps     : tps;
```

```
      str    : tstr;
      ar1    : tarr;
```

```
      keys   : tKey;
```

```
      str:= '1234';
      ar1[3]:= 56;
```

```
      pb:= tpb (pa);
      pb^:= 0;
```

```
      keys:= Stop;
```

## 4.6.11 CONST

Starting of a constant declaration

The constant definition part introduces identifiers as synonymous for constant values. The reserved word **const** heads the constant definition part, and is followed by a list of constant assignments separated by semicolons. Each constant assignment consists of an identifier followed by an equal sign and a constant. Constants can be of any scalar type, strings, records or arrays.

Please note that the scalar constants are **not** stored into the ROM/FLASH while constant arrays, strings and records are always stored into the Flash.



# AVRco Compiler-Manual

## 4.6.11.1 Predefined Constants

The following constants are predefined and may be referenced without previous definition:

<u>Name:</u>	<u>Type and values:</u>
<i>False</i>	Boolean (boolean value false).
<i>True</i>	Boolean (boolean value true).
<i>Pi</i>	Float pi
<i>Nil</i>	Zero, also zero value pointer.

The following predefined constants have a **special construction**: they dynamically generate a string which is constructed of the actual Computer Date or Time:

<i>Date</i>	current date as a string constant into the ROM
<i>Time</i>	current time as a string constant into the ROM

### **Const**

```
Date = 'dd.mm.yy';    // -> '26.12.99'  
Date = 'dd.mm.yyyy'; // -> '26.12.1999'  
Date = 'mm/yy';      // -> '12/99'  
Time = 'hh:mm:ss';   // -> '22:02:06'  
Time = 'hh:mm';      // -> '22:02'  
Time = 'hhmm';       // -> '2202'
```

## Compiler Version

The compiler version can be found with predefined constants and can be used within the program.

```
CompilerRev   : word = rev;      // actual Compiler version  
CompilerBuild_Y : byte = yy;     // actual Compiler build, last 2 digits of current year 00..99  
CompilerBuild_M : byte = mm;     // actual Compiler build, current month 01..12  
CompilerBuild_D : byte = dd;     // actual Compiler build, last current day 01..31  
CompileYear    : byte = yy;     // last 2 digits of current year 00..99  
CompileMonth   : byte = mm;     // current month 01..12  
CompileDay     : byte = dd;     // current day 01..31  
CompileHour    : byte = hh;     // current hour 00..23  
CompileMinute  : byte = mi;     // current minute 00..59  
ProjectBuild   : word = pbuild;  // number, incremented with each successful project compile
```

## 4.6.11.2 Type Specification with Constant Declaration

With Borland Pascal constants have to be defined without a type declaration:

```
const abc = 1;
```

The value "1" now can be used as byte, word, integer, float etc. This is similar in the AVRco Pascal. But sometimes there are problems with such ambiguous values. Because of this the ordinal constants should be defined like this:

```
const abc : byte = 1;
```

See also Compiler Switch `{$TYPEDCONST ...}`

This fixes the value of a constant to a specific type, in the example above it is an unambiguous byte.

### Warning:

in Borland the meaning of this construction is always a structured constant. Do not confuse the two kinds!

### 4.6.11.3 Constant Load from File

Array and Record constants can be read in from a file. The programmer is responsible of the contents of the file. The file length is not important. If the file is too short, the structure is filled with zeros. If the file is too long, the read-in is aborted at the Array/Record-limits.

```
const name : array[a..z] of byte = 'FileName.ext';
```

This operation is also valid for records where the record must be a predefined type. Such file constants can also be placed into the StructConst area of the EEPROM.

As an option it is possible that the filesize itself defines the array size.

```
const name : array = 'FileName.ext';
```

The array now is defined as **array**[0..filesize-1] **of** byte.

```
const FileConst : array = 'E-LAB.pbmp';
```

It is possible to add an optional type to a constant which is loaded from a file. This type must be a simple one (Byte...Float). The Array now is implemented in this manner:

```
const array[0..(filesize div sizeOf (type))-1] of Type.  
const FileConst : array of Char = 'E-LAB.txt';
```

Constant Records and Arrays can be read out of more than one file for such constants. They will be read sequentially.

```
const ArrXYZ : tArrXYZ = 'FN1.ext', 'FN2.ext', 'FN3.ext';
```

The filenames must be separated by commas. The files are read in until the structure is full. The rest will be ignored.

If the sum of the file bytes is smaller than the structure size the rest of the structure is filled with zeros.

#### **Example:**

An Example can be found in the directory **..E-Lab\AVRco\Demos\LCD\_PCF8548**

### 4.6.11.4 Constant Located in Flash

Byte and Word constants can be placed into the Flash at fixed addresses:

```
{ $PHASE $01EFF }  
ASM;  
.WORD 0AA55h  
ENDASM;  
{ $DEPHASE }
```

The switch \$PHASE defines the absolute start address of the following statement block.

This address is a **WORD** address.

The address \$01EFF above results in an absolute Byte-address of \$03DFE.



# AVRco Compiler-Manual

It's possible to place ordinal, Array and Record constants into dedicated addresses in the Flash. This is done as with variables

```
const wwC[$1000] : byte = 123;  
      fltC[$1010] : float = 0.5;
```

The address follows the name/identifier in []. A type (byte, word etc) is necessary. With arrays and Records a previously defined type must be used. The address is only checked for Flash boundaries but not for plausibility. Use absolute addresses with care!

Further it's possible that the compiler generates the placement like it does with variables:

```
const  
  Name[@FLASH] : type = value;
```

The identifier "FLASH" forces the automatic address generation and placement.

All constant definitions like:

```
const  
  name   = 'x';  
  name1  = 'xxx';
```

are placed into the ROM as string constants.

If character literals (immediate constants) must be possible, which are not placed into the ROM, these must be qualified like this:

```
const  
  name   : char = 'x';  
  name1  : char = #12;
```

It is possible to place constants into ROM/FLASH which contain Pointer which also point into the ROM/FLASH. You find an example in the directory **..E-Lab\AVRco\Demos\Mega161**.

Please note that predefined/user defined arrays and records must always be used. These types must be defined in the type declaration part of the application.

With all constants in ROM/FLASH the following is always valid:

If the name of the constant never appears in the context (never referenced) the optimizer removes this constant. If only a pointer is used to reference such a constant and the pointers value is build at runtime, the constant is probably removed. If so, an assembler error is generated. To avoid this, one can make a dummy access onto this construct by its name.

Types, constants and variables should be declared completely before the first procedure or function. Most of the calculations and operators are also permitted for constant declarations.

Complex constants (Arrays and Strings) are stored in ROM (Rom constants). It is therefore necessary for the programmer that corresponding ROM space is allocated and these values are not normally alterable (read only).



```

const x1 = Lo (1234);
        x2 = Hi (1234);
        x3 = 24 * 2;
        x4 = x2 div 2;
        x5 = 12 mod 10;
        x6 = 1 shl 4;
        x7 = 1200 or 34;
        x8 = $FFFF and $AAAA;
        str0 = '1234' + 'R' + #7 + ^L;
        str1 = '1234';
        ch = '9';
        TAB = ^\;
        TB = TAB;
        bits = 3;
        st = 'Hallo';
        x9 = %11001101;
        arr : array[3..7] of byte = (0, 1, 7, 34, 128);
    
```

## 4.6.12 STRUCTCONST

structured Constants

```

ars      : array[3..7] of byte = (0, 1, 7, 34, 128);
px       : pointer to word = $40;
sc[$15] : word = $1234;           {fixed addr $15}
    
```

Structured constants behave like variables at run-time, that means they possess an address and they are readable and writeable. The value and its address, which are treated as 'constant' are stored in the program (ROM) or EEprom. After the memory initialization the stored values of the constants are loaded from ROM into RAM to the corresponding address and are now able to be handled as a variable. Because a structured constant possesses an address, a fixed address can also be optionally assigned by using the syntax `xx [adr] : byte = bb; .`

A destination memory area as a compiler switch should always be prefixed before a StructConst declaration, e.g. `{$IDATA}`.

A special case is the use of structured constants in EEprom, if EEPROM is available in the processor. The declaration of such a constant looks like this:

```

{$EEPROM}
Structconst
    ee1 : word = $1234;
    est : string = 'hello';
    eft : float = 1.23456;
{$IDATA}
    
```

All structured constant declarations following the compiler switch **EEPROM** now are going to the EEprom of the CPU (until switched back again e.g. with `{$IDATA}` ). These constants stay in the EEprom and will not be copied into RAM. Instead the compiler generates a special Hex-File which has the ending `xxx.eep`, where these constants are held.

The programmer-tools are generally able to read a special Hex-file for the On-chip EEprom and they are able to program corresponding the CPU. The EEprom is programmed with the assigned values.

The defined constants are in the EEprom and can be read and written like normal variables with help of the CPU specific addressing algorithms, which are used automatically by the compiler.

## Special construction

Array and Record constants can be read in from a file. The programmer is responsible of the contents of the file. The file length is not important. If the file is too short, the structure is filled with zeros. If the file is too long, the read-in is aborted at the Array/Record-limits.

**Structconst** *Arr1* : **array**[0..31] **of** *word* = 'FileName.ext';

## 4.6.13 VAR

Starting of the variable declaration

Every variable occurring in a program must be declared before use. The declaration must textually precede any use of the variable so that the variable is 'known' to the compiler when it is used.

A variable declaration consists of the reserved word **var** followed by one or more identifier(s), separated by commas, followed by a colon and a type. This creates a new variable (or variables) of the specified type and associates it with the specified identifier.

The 'scope' of this identifier is the block (function, procedure) in which it is defined, and any block within that block. This variable is said to be *local* to the block in which it is declared, and the variable access from the outer level becomes impossible. In Pascal the search for an identifier starts from the inner block and ends global. This means within a Procedure/Function/Unit the search starts with local definitions. If nothing found the search continues with global definitions, which also includes the definition part of Units.

Types, constants and variables **should be** declared **before** the first procedure or function. All variables require appropriate memory area. For the respective requirements read **types** above. Variables, which are not required by the program only waste the limited memory. So a careful handling with these declarations is necessary. (see also compiler switch \$W)

Generally all variables have to be stored in a certain memory page of the CPU. The possible pages are : **Data**, **iData**, **iData1**, **pData**, **xData** and **EEprom**, **EEprom1**, depending on the type of processor used (File xxx.dsc). The actual page is selected by the according compiler switch , e.g. **{SIDATA}**. Pay attention to capitalisation! This memory page is preserved until the next re-definition.

Note that the last selected page is valid for the main program and also for some system variables. The **Default value** of the term 'Implementation' is, if it exists, 'Data', or if not 'iData'.

A **global** variable can with an attached address be stored at a fixed memory location. This is necessary for memory-mapped ports and control registers. Here must be observed that the assigned address also is in the actual memory page (Data, iData, pData, xData, EEprom). Variables without addresses are stored by the compiler at the next available memory location in the currently selected memory page.

```
{ $DATA }
Var TrisA[$85]      : byte;           {var at fixed addr.}
      TrisB[$86]      : byte;           {var at fixed addr.}
      Count           : byte;           {normal declaration}
      i               : integer;
      ix[@j]          : byte;           {Lo-byte of i}
      iy[@i + 1]      : byte;           {Hi-byte of i}
      Bit7[@TrisB, 7] : bit;            {bit7 in TrisB Reg}
      Ar1             : array[2..9] of byte; {array}
      St1             : string[6];      {string of length 6}
```

## 4.6.14 LOCKED

**Global** ordinal variables can be extended by the attribute **locked** in order to protect them against concurrent accesses of interrupts, processes or tasks. For example if two processes have access to a global Bit-var, it's possible that a process starts a read-modify-write access with **Incl(Bit)** and after the Bit-read (byte access) is interrupted by the other process. Now this process changes this byte which the first process has read before. After passing the control back to the first process this one writes back the manipulated bit (byte-write). This write access overwrites the changes made by the second process. **Locked** prevents this by disabling a Task/Process change or interrupt during the access to such protected vars.

Read-Modify-Write statements are : INC, DEC, INCL, EXCL, SETBIT.

With 8-Bit processors also read or write accesses to global vars, which are greater than a byte, should be protected if concurrent access is possible. The **Locked** attribute should be used carefully and rare because each access generates an overhead of code and cycles. Also the interrupt is disabled during the whole access.

```
{ $IDATA }
Var i : integer, locked;    {protected}
      Bit9[@i, 9] : bit;     {bit9 in i}
```

The integer variable **i** is locked and protected and also the Bit-variable **Bit9**, which is derived from **i**.

### **Attention:**

Continuously polling or read of a "locked" variable e.g. within a loop can lead to a distortion of Task/Process-changes, because interrupts are disabled nearly all of the time.



# AVRco Compiler-Manual

**Local** variables within a procedure are stored in an area known as the frame and they are only accessible within this procedure because the frame and the variables only exist temporarily. Local variables force an indexed address calculation and a read/write access with a pointer. That leads to an increase of code and runtime.

Local variable are not initialized (default). See compiler switch `{$ZeroLocVars}`

**Procedure** *LocalVars*;

**Var** *bb* : Byte;                    {local variable}

**begin**

*bb:= not bb;*

**end;**

**Process** and **Task** variables are programmed like normal local variables, but they always must exist.

So they are static and will be directly addressed like global variables. In spite of this they are not global but encapsulated, i.e. they are not normally accessible from outside the process in which they are defined.

However if it is necessary manipulate them from outside the process, the process name has to be prefixed before the variable,

e.g. *procName.var1:= 0;*

With concurrent accesses by other Processes or Tasks these variables must get the "locked" attribute from the user (no automatic locking).

{ *process-vars and process-stack into xData*}

**Process** *DoTheJob* (20, 10 : *xData*);

**Var** *pb* : Byte;                    {local var, but static }

**begin**

*bb:= not bb;*

**end;**

{*\$IDATA Main-stack into iData*}

**begin**                                {Main}

*DoTheJob.pb:= \$FF;*

**end;**

## 4.7 Procedures and Functions

A Pascal program consists of one or more *blocks*, each of which may again consist of blocks. One such block is a *procedure*, another is a *function* (commonly called *subprograms*). Thus, a procedure is a separate part of a program, and it is activated from elsewhere in the program by a *procedure statement*. A function is rather similar, but it computes and returns a value when its identifier, or *designator*, is encountered during execution.

C programmers are familiar with the type function. A procedure is a function that does not return any result. This may seem unnecessary but it actually allows the compiler to produce more compact code.

### Parameters

Values may be passed to procedures and functions through *parameters*. Parameters provide a substitution mechanism which allows the logic of the subprogram to be used with different initial values, thus producing different results.

The procedure statement or function designator which invokes the subprogram may contain a list of parameters, called the *actual parameters*. These are passed to the *formal parameters* specified in the subprogram *heading*. The order of parameter passing is the order of appearance in the parameter lists. Pascal supports two different methods of parameter passing, by *value* and by *reference*, which determines the effect that changes of the formal parameters have on the actual parameters.

When parameters are passed *by value*, the formal parameter represents a local variable in the subprogram, and changes of the formal parameters have no effect on the actual parameter. The actual parameter may be any expression, including a variable, with the same type as the corresponding formal parameter. Such parameters are called a *value parameter* and are declared in the subprogram heading as in the following example. This and the following examples show procedure headings; function headings are treated later.

*procedure Example (Num1, Num2 : Number; Str1, Str2 : Txt);*

*Number* and *Txt* are previously defined types (e.g. *Integer* and *string[255]*), and *Num1*, *Num2*, *Str1*, and *Str2* are the *formal parameters* to which the value of the *actual parameters* are passed.

The types of the formal and the actual parameters must correspond.

Notice that the type of the parameters in the parameter part must be specified as a previously defined *type identifier*. Thus, the construct:

*procedure selectModel (array[1..5] of Integer);*

is not allowed.

Instead, the desired type should be defined in the **type** definition of the block, and the *type identifier* should then be used in the parameter declaration.

**type**

Range: *array[1..5] of integer;*

*procedure Select (Model: Range);*



# AVRco Compiler-Manual

When a parameter is passed *by reference*, the formal parameter in fact represents the actual parameter throughout the execution of the sub program. Any changes made to the formal parameter are thus also made to the actual parameter, which must therefore be a *variable*. Parameters passed by reference are called a *variable parameters*, and are declared as follows:

**procedure** Example (*var* Num1,Num2: Number)

Value parameters and variable parameters may he mixed in the same procedure as in the following:

**procedure** Example (*var* Num1, Num2 : Number; Str1, Str2 : Txt);

in which *Num1* and *Num2* are variable parameters and *Str1* and *Str2* are value parameters.

All address calculations are done at the time of the procedure call. Thus, if a variable is a component of an array, its index expression(s) are evaluated when the subprogram is called.

When a large data structure, such as an array, is to be passed to a sub program as a parameter, the use of a variable parameter will save both time and storage space, as the only information then passed on to the subprogram is the address of the actual parameter. A value parameter would require storage for an extra copy of the entire data structure, and the time involved in copying it.

## **Tip:**

Make your job easier and improve the readability of your applications by an unambiguous use of functions:

**Function** ReturnAbyte : byte;

**begin**

...

**end;**

This function then is usually used in this manner:

*var* bb : byte;

...

bb:= ReturnAbyte;

But looking at the statement it's not clear whether ReturnAbyte is a variable or a function.

But if you write

bb:= ReturnAbyte();

it's clear, it's a function.

With the following constructions the empty parenthesis is a must.

**Function** ReturnApointer : pointer;

**begin**

...

**end;**

...

bb:= ReturnApointer()^;

ReturnApointer()^:= bb;

Here the function result (pointer) is directly used to implement a move.

## 4.7.1 PROCEDURE

### Procedure Declaration

A procedure declaration serves to define a procedure within the current application or Unit. Procedures within procedures are not allowed. A procedure is initiated from a procedure statement, and upon completion, program execution continues with the statement immediately following the calling statement.

A procedure declaration consists of a procedure heading followed by a block which consists of a declaration part and a statement part.

The procedure heading consists of the reserved word **procedure** followed by an identifier which becomes the name of the procedure, optionally followed by a formal parameter list as described earlier.

All identifiers declared in the formal parameter list and the declaration part are local to that procedure. This is called the *scope* of an identifier, outside which they are not known. A procedure may reference any constant, type, variable, procedure, or function defined in an outer block.

The statement part specifies the action to be executed when the procedure is invoked, and it takes the form of a compound statement. If the procedure identifier is used within the statement part of the procedure itself, the procedure will execute recursively. This may be dangerous for inexperienced programmers.

Procedures are sub-routines, which can be called by name. There is a distinction between parameterless procedures and procedures with parameters.

Example for a parameterless procedure:

```
Procedure Test1;  
begin  
  Statement ...;  
  Statement ...;  
end;
```

Example for a procedure with parameters:

```
Procedure Test2 (par : byte);  
begin  
  if par > 0 then  
    ...  
  else  
    ...  
  endif;  
end;
```

The passing parameter (argument) is, if there is no **var** prefixed, generally an invariant, i.e. it is possible to manipulate it within the procedure. It can be changed, but the original (at the calling location) is not changed, because the parameter is only a copy of the original.

Often the changes made to the formal parameters in the procedure should also affect the actual parameters.

If a **var** is prefixed the original variable is used, not a copy! The parameter is treated like a pointer.

```
Procedure Test2(var par : byte);
```



# AVRco Compiler-Manual

Passing parameters are passed by the frame-stack and stay on the frame. That leads to a bigger code and a slower execution speed than with the global variable. If a procedure/function is time-critical, you should work without passing parameters and local variables.

Recursions (e.g. calling of *Test1* within the procedure *Test1*) are certainly allowed, but they could quickly lead to stack problems (e.g. overflow).

**Local variables** in functions and procedures have to be considered from the same point of view as passing parameters, namely a bigger and slower code.

```
Procedure Test3;  
var loc : boolean;  
begin  
  if loc then  
    ...  
  endif;  
end;
```

Every procedure **is able to** be completed (exited prematurely) with *Return*.

```
Procedure Test1;  
begin  
  Statement ...;  
  if (a > b) then  
    Return;  
  endif;  
  Statement ...;  
end;
```

## 4.7.2 PROCEDURE SYSTEM\_INIT

Special Procedure Declaration

System\_Init is executed after the stack-initialisation. The user program is here able to execute certain hardware-initialisation, before the system is doing its own initialisation. After the procedure follows the system internal memory-, hardware,- and structured constant-initialisation etc.

```
Procedure System_Init;  
begin  
  Statement ...;  
  Statement ...;  
end;
```

### Attention:

The program itself must not call the System\_Init!!



## 4.7.3 PROCEDURE SYSTEM\_MCUCR\_INIT

The mega128 supports different WAIT states at some external memory areas. Furthermore parts of the address bus ports can be used as standard IOs.

In order to handle this the MCUCR register must be accessed immediately after a PowerOn or reset.

If this callback function is present in the application the system internal initialization of the MCUCR is omitted and now is completely under control of the application.

In order to do this the application must provide the following procedure:

**Procedure** *SYSTEM\_MCUCR\_INIT*;

**begin**

*MCUCR:= bb;*

**end;**

At this time no other initialization has been done. So local vars, calling Library functions etc. are forbidden.

## 4.7.4 FUNCTION

Function Declaration

A function declaration consists of a function *heading* and a *block* which is a declaration part followed by a statement part. The function heading is equivalent to the procedure heading, except that the heading must define the *type* of the function result. This is done by adding a colon and a type to the heading.

The declaration part of a function is the same as that of a procedure. The statement part of a function is a compound statement as described with procedures.

A function declaration serves to define a program part which computes and returns a value. A function is activated when its designator is met as part of an expression. It is also possible to call a function like a procedure. In this case the result is discarded.

Functions are sub-routines, which are called by name. A function generally returns a result. The *Return*-statement is a **must** and has to be added with the result of the function. In general the same principles apply as with **Procedure**. The distinction between parameterless functions and functions with parameters are as with **procedures**.

Example for a parameterless function:

**Function** *Test1* : *boolean*;

**begin**

*Statement ...;*

*Statement ...;*

**Return**(*a > b*);

**end;**

Example for a function with parameters:

**Function** *Test2* (*var par* : *byte*) : *byte*;

**begin**

**if** *par > 0* **then**

**Return**(0);

**else**

**Return**(1);

**endif;**

**end;**



# AVRco Compiler-Manual

Example for a function with local variable:

```
Function Test3 : boolean;  
var loc : boolean;  
begin  
  if loc then  
    Return(loc);  
  endif;  
end;
```

With the use of the Compiler switch {\$NORETURNCHECK} it's possible to omit the return statement in a function. This is only for special purposes.

The result of a function can also be an array or record. But the result of such a function must always be assigned to a variable of the correct type.

Other assignments or the usage of the result in an expression are not possible.

```
type tRec = record  
  abc : byte;  
  ...  
end;
```

```
var rec : tRec;
```

```
function GetRec : tRec;  
begin  
  return(rec);  
end;
```

```
...  
rec:= GetRec();    // legal
```

Not possible is:

```
GetRec().abc:= $12; // illegal  
xy:= GetRec().abc;  // illegal
```

## 4.7.5 PROCESS

Process Declaration

```
Process ProcessName (StackSize, FrameSize : word; DataPage);
```

Processes are independent programs in an application, which are able to run completely independent from other program parts (e.g. main). Processes can not be called like procedures or functions. They are instead called periodically by what is known as a scheduler, based on their priority. If processes were imported, the main program also runs as a process and has a priority, too.

If several processes exist in a program, the processing of the separate processes operate quasi-parallel, i.e. considered from outside all tasks/processes seem to work at the same time = **Multi-Tasking**. So a pseudo **parallel-processing** is achieved for events and data, although they are not really executed simultaneously.

In practice a process runs endlessly, only interrupted by interrupts and other processes and tasks. The 'begin' and 'end' limits a process with respect to its statements. Because processes cannot be called like functions they cannot have passing parameters and, may return no results.

On the first call by the scheduler it starts with the statement which comes after the 'begin'. Now all statements will get processed until 'end', eventually interrupted by a **task switching** through the scheduler (switching to another process/task). If 'end' is reached, it is automatically continued with the first statement after 'begin'. So a process runs continuously in a circle respective without an end. But the programmer has not to program a loop, because the jump to the begin is automatically. Here is the essential difference to a task. Tasks break up with 'end' and pass the control to the scheduler respective to the next process.

To declare a process, the stacksize and the framesize (10..1000 bytes), which are required by this process and the data area (iData, xData etc) has to be specified. The stacksize is depending on the depth of the sub-routines calls, the according address pushes, and the parameter pushes caused by various statements. The framesize is only depending on the frames (local and passing parameters) of the called sub-routines. The exact required amount of bytes is in practice hard to establish because all eventualities must be traced by a debugger. It is not the size (number of statements) of the processes that is relevant, but the kind of the statement (for example Floating Point or multiple nested sub-program calls). If there is enough ram available, then 'the more the better'. An acceptable value for fairly simple processes is 32 bytes for the stack and 16 bytes for the frame.

Every process requires ca. 20 byte memory, where during an interruption by the scheduler the **pseudo-accumulators** (working registers), stack pointer and flags of the process are stored.

**Local variables** within processes and tasks are indeed encapsulated i.e. there is no direct access from outside, but they are located in the normal memory like global, static variables. Local variables must be available and there must always be a possibility of access. It makes no difference if the process is active, asleep or is suspended. So they reside in normal memory in ram and they have to be included in the calculation of the required memory. The access within the process or the task is made by the declared name. From outside (other processes, main etc) an access can be made by using the qualified name '*ProcessName.VarName*'.

The above mentioned memory areas (Register-backup, stack, frame and local variable) are within the memory page, which was specified by the process declaration (\$IDATA, \$XDATA etc).

If processes/tasks are imported, the **main program** is treated as a process, so a compiler switch should be used to select the desired memory area immediately before the **begin**.

```
Process DoTheJob (32, 16 : iData); {Stacksize = 32 bytes, Framesize = 16 bytes}  
var px : integer;  
begin  
  Statement ...;  
  Statement ...;  
end;
```

The method of working is controlled by a large number of relevant functions and procedures. An essential parameter is priority.

With **priority** a part of the available run-time is placed at disposal to a process. The higher the value of priority is, the more run-time is at disposal. At the same time priority predefines the number of system ticks, which are completely available for the process in one time slice. The proportional runtime of the total time in % is calculated by: Priority / Sum of all priorities.

Assumed there is only the process 'DoTheJob' and it has the priority 10 and Main Priority is 5 then % time available for the process is (estimated):  $\text{run-time} = 10 / (5 + 10) = 66\%$ . The exact run-time can only be calculated if no process is suspended or locked and if no ProcessWaits etc. exist. In practice the proportional run-time can only be estimated.



# AVRco Compiler-Manual

A process/task is able to take over the CPU completely by **lock**, so apart from itself only interrupts are running. This state is cancelled by **unlock**.

If a process/task establishes that it has nothing to do at the moment, there should be no waste of run-time by using waitloops or delays. There are several ways to pass the control to other processes:

With **Schedule** the process/task is interrupted immediately, but is kept in the list of active processes.

With **Sleep** a process/task is able to switch itself off for a certain number of system ticks.

With **Suspend** a process/task switches off. It is not able to switch itself active again. This must be done from another process/task or by the main program using the **resume** procedure.

Because the communication between tasks/processes is made by pipes and semaphores, there is also the possibility that the task switches itself off by calling **WaitSema** or **WaitPipe**. The process/task becomes active if there is data in the specified semaphore or pipe. It is also possible for **RxBuffer** to be specified as a pipe.

The process/task is interrupted directly after an above mentioned instruction.

## 4.7.5.1 Define Options

It is possible to optionally define a Process's priority and/or initially suspend or resume it:

*Process Name (StackSize, FrameSize : MemoryArea[; Priority, RunMode]);*

```
Process Proc1 (32, 32 : iData);           // default prio=3, autostart  
Process Proc1 (32, 32 : iData; 5);       // priority 5  
Process Proc1 (32, 32 : iData; resumed); // default prio=3, automatic start  
Process Proc1 (32, 32 : iData; 5, suspended); // prio= 5, no automatic start
```

## 4.7.6 TASK

Task Declaration

*Task TaskName (DataPage);*

Tasks are independent programs within an application, which are able to run absolutely independently from other program parts (e.g. main), i.e. tasks cannot be called like functions or procedures. Instead they are called periodically by the scheduler. If tasks have been imported, the main program runs as a process, and also has a priority. Tasks are **extremely specialized processes** and they only should be used for certain jobs, for example PID-controller.

If there are several processes/tasks within a program, the tasks are done quasi-parallel, i.e. it seems like all processes are done at the same time = **Multi-Tasking**. So an apparent **parallel-processing** for example of events or data is achieved, although they do not actually run simultaneously.

In practice a task runs perpetually, only interrupted by interrupts and other processes and tasks. The 'begin' and 'end' limits a task with respect to its statements. Because tasks can not be called like functions, they do not have any passing parameters or results.

With **every** call of a task by the scheduler, it is started with the statement which immediately follows the 'begin' statement. Following statements are all processed until the 'end' statement is reached. If the 'end' is not reached within a system tick, the task is **interrupted** by a **task change** by the scheduler (Switch over to another process/task). So the task never reaches the 'end', if its required run-time from 'begin' to 'end' is longer than a system tick. The run-time must **never** be longer than a system tick. Similar conditions are also valid for interrupts. A timer-interrupt-service-routine for example, should never take more time than the period between two interrupts.

If the 'end' is reached, the control is automatically passed to the scheduler, which now activates the next process or task. In contrast to a process a task runs from 'begin' to 'end' and then aborts itself.

If tasks have been imported, a **TaskStack** and a **TaskFrame** have to be defined. All tasks within the system use the same stack and frame, i.e. only one task stack and frame exists, because tasks are normally not interrupted by the scheduler and always resume at the same point (statement). The required **stack size** is determined by the demand of the largest task (7..255 bytes). The stack size depends on the depth of the calls of sub-programs and the resulting address push's as well as on parameter pushes, caused by various statements. In practice the exact number of required bytes can not be established, because all eventualities have to be traced.

As with all other stack definitions it has to be estimated with common sense and experience. It is not the size (number statements) of the task is important, but the kind of statement (for example Floating Point or multiple nested sub-program calls). If there is enough ram available, the best policy is 'the more the better'. An acceptable value for a simple task is 32 bytes.

Every task requires ca. 20 byte memory, where during an interruption by the scheduler the **pseudo-accumulators** (working registers), stack pointer and flags of the process are saved. This memory is also the same for all tasks and exists only once.

**Local variables** within processes and tasks are encapsulated, i.e. there is no direct access from outside, but they are held in normal memory like global and static variables. Local variables must be available and there must always be a possibility of access. It makes no difference if the task is active, asleep or is suspended. They are usually use memory in ram and they have to be included in the calculation of the required memory. Access within the process or the task is made by the declared name. From outside (other processes, main etc) an access can be made by 'TaskName.VarName'.

The above mentioned memory areas (register-backup, frame and stack) are within the memory page specified by Define TaskStack = size, RAMpage (iData, xData etc) and Define TaskFrame = size. The 'local' variables of a task also reside in this memory page.

```
Task RunPid (xData);  
var tx : integer;  
begin  
    Statement ...;  
    Statement ...;  
end;
```

The method of working of a task is controlled by a large number of relevant functions and procedures. An essential parameter is priority.

As opposed to a process, the calling interval of the **task** is predefined by **priority**. The lower the value of priority is, the more often the task is called. Assumed the task 'RunPid' has the priority 10, then it is called every 10 Systicks.

Thus the period between two calls is always 10 ticks.



# AVRco Compiler-Manual

## **Note:**

If there are several **tasks**, and if it is possible, that several of them are active, pay absolutely attention that all priorities have a **common denominator**.

I.e. all priorities must be a multiple of 2, for example. If this condition is not met, then there are irregular call intervals, i.e. the period between two calls is not constant any more. Further the lowest priority has to be higher than the sum of all tasks.

If a task establishes that it has nothing to do at the moment, there should be no waste of run-time by waitloops or delays. There are several possibilities to pass the control to other processes:

With **schedule** the task is interrupted immediately, but is enqueued again into the waiting loop.

With **sleep** a task is able to switch itself off for a certain number of system ticks.

With **suspend** a task switches off. It is not able to switch itself active again. This must be done by a task from outside or by the main program with **resume**.

The task is directly suspended after any instruction mentioned above.

A task is able to take over the CPU completely by **lock**, so that except itself only interrupts are running. This state is abolished by **unlock**.

Because the communication between tasks/processes can be with pipes or semaphores, there is the possibility that the task switches off by calling **WaitSema** or **WaitPipe**. The task gets active again if there is data in the specified semaphore or pipe. **RxBuffer** can also be specified as a pipe.

## 4.7.6.1 Define Options

It is possible to optionally define a Task's priority and/or initially suspend or resume it:

**Task Name** (*MemoryArea* [*;* *Priority*, *RunMode*]);

```
Task Task1 (iData);           // default prio=5, autostart
Task Task1 (iData, 8);       // priority 8
Task Task1 (iData, resumed); // default prio=5, automatic start
Task Task1 (iData, 8, suspended); // prio= 8, no automatic start
```

## 4.7.7 FORWARD

Forward declarations of pointers, functions, procedures and processes

A forward reference is properly not possible in Pascal as well as in other programming languages and some assemblers. I.e. all statement referenced elements (types, variables, constants, procedures, functions) have to be declared before they could be used.

Sometimes it is absolutely necessary that a procedure/function/process is referenced before it has been declared. Therefore a *Forward* declaration is possible. The procedure/function/process head is written again after the variable/constant declaration and before the first function/procedure declaration with the addition 'forward'. The proper declaration can happen at any later position. Here *forward* must not be used again!

```
Program Abc;  
...  
var ...;  
const ...;  
  
Procedure Test1; Forward;  
Process Proc1(32); Forward;  
Funktion Test2(par : byte) : byte; Forward;  
...  
...  
Process Proc1(32);  
begin  
...  
end;  
  
Procedure Test1;  
begin  
  Statement ...;  
  if (a > b) then  
    Return;  
  endif;  
  Statement ...;  
end;  
  
Funktion Test2 (par : byte) : byte;  
begin  
  if par > 0 then  
    Return(0);  
  else  
    Return(1);  
  endif;  
end;
```

*Forward* also can be used for type declarations, particularly for pointers.

```
Type TPtr = pointer; forward;  
  TRec1 = record  
    ...  
    Ptr1 : TPtr;  
  end;  
TPtr = pointer to Trec1;
```



## 4.7.8 BEGIN

Starting procedure-, function-, process- or task body

Every procedure, function, process, task as well as the main program has to begin with a **begin** and to end with an **end**; as you see also in above mentioned examples. Pascal programmers will surely have already recognized that after a *then* and *else* generally no begin and no end statement follows.

The designer of Pascal, N.Wirth, has not been consistent by this and the result is that begin/end is either a *must* or a *can*. A concrete example is the exception with the *Case..Else* statement. Every programmer must often consider, how did it come about? In the successor of Pascal, *Modula-2*, Wirth clarified it, and he abolished the begin/end in these constructs. This made the relevant compiler construction easier and safer at this point.

This feature (as well as others) has been introduced in the existing compiler. The protest: This is not now Pascal-compatible, can be ignored in a system, which knows as many hardware dependences as an 'embedded' development system. To make development easy there have to be many language extensions introduced.

Anyone who knows Borland-Pascal or Delphi, also knows that about half of the construction is not standard Pascal compatible either.

### Statements

The statement part is the last part of a block. It specifies the actions to be executed by the program.

The statement part takes the form of a compound statement followed by a period or a semicolon.

A compound statement consists of the reserved word **begin**, followed by a list of statements separated by semicolons, terminated by the reserved word **end**.

The final **end** in a program is followed by a "." (dot) in traditional Pascal.

## 4.7.9 RETURN

Abort and exit within a procedure/function

The *Return* statement allows for the abort (return) of a procedure or a function at any point. Return is a **must** for all functions. (With the use of the Compiler switch {\$NORETURNCHECK} it's possible to omit the return statement in a function, but this is only for special purposes.)

Every procedure **can be** terminated with the *Return* statement. The Return statement is parameterless.

### *Procedure Test1;*

**begin**

*Statement ...;*

**if** (*a > b*) **then**

**Return;**

**endif;**

*Statement ...;*

**end;**

Every function **has to be** terminated by the *Return*-statement.. Return **has to have** a parameter. The type of this parameter has to be declared in the function header. Permissible types are 8, 16, 32 and 64bit values.

### *Function Test2 : boolean;*

**begin**

*Statement ...;*

**Return**(*a > b*);

*Statement ...;*

**end;**



## 4.7.10 END

End of a procedure-, function-, task- or process-body

Every procedure, function, task and process **has to** begin with a ***begin*** and end with an ***end***; see also examples above. The end-statement has to be followed by a semicolon. See also description of *Begin*. The *End*-statement is in contrast to standard Pascal only permitted for the end of functions procedures and the main program. IF, WHILE, REPEAT etc. have their own qualified abort, i.e. ENDIF etc. This is for a better readability of the source.

For a better readability it is optionally possible to insert the name of the procedure/function in front of the semicolon ( ***end Test2;*** )

## 4.7.11 ASM:

Start of a single Assembler statement in a Pascal source.

***ASM: PUSH\_ACCA;***

## 4.7.12 ASM;

Start of an assembler text block

A program for Embedded Control often does not work without assembler code, because either the compiler generated code is too slow for some operations, or certain assembler commands have to be done, which are not known or not used by the compiler.

Assembler source can be inserted directly at every position of the Pascal source. This source is passed untested and unprocessed by the compiler to the assembler. Because the compiler also generates assembler code, the assembler text is seamlessly inserted.

Asm-syntax errors are only recognized by the assembler, not by the compiler.

Access to all declared variables is possible in the assembler text.

### **Attention:**

Labels in an assembler-block have to start at the beginning of a line and terminated with a ':'. This line may not contain any further instructions, e.g. code.

The analysis of the compiler generated assembler-files 'xxx.ASM' may help:

### ***ASM;***

*LDI* *\_ACCA, 040h*

*ANDI* *\_ACCA, myProg.a;*            *{a = Pascal var in myProg }*

### ***ENDASM;***

## 4.7.13 ENDASM

End of an assembler text

ASM and ENDASM are Pascal statements and have to end with a semicolon ;



## 4.8 INTERRUPTS, TRAPs and EXCEPTIONs

### 4.8.1 INTERRUPT

Declaration of an Interrupt Procedure.

Possible interrupt sources totally depend on the processor. Within a CPU-family there are considerable differences. Interrupt sources are defined declared in the processor description file (xxx.dsc).

This procedure sets only an entry in the interrupt-vector-table and creates a program frame for the selected register savings (see below).

**In addition the I/O control registers of the CPU for the specific interrupt handling must be initialized by the application itself (see the descriptions in the controller manual) !**

**Example** (AVR Mega103):

<b>TIMER0</b>	timer0 overflow interrupt
<b>TIMER0COMP</b>	compare match interrupt timer0
<b>TIMER1</b>	timer1 overflow interrupt
<b>TIMER1COMP</b>	compare match interrupt timer 1
<b>TIMER1COMPA</b>	compare match "a" interrupt timer 1
<b>TIMER1COMPB</b>	compare match "b" interrupt timer 1
<b>TIMER1CAPT</b>	capture event timer 1
<b>TIMER2</b>	timer2 overflow interrupt
<b>TIMER2COMP</b>	compare match interrupt timer2
<b>EERDY</b>	eeprom ready
<b>ACOMP</b>	analog comparator
<b>SPIRDY</b>	SPI serial transfer complete
<b>ADCRDY</b>	ADC conversion complete
<b>INT0</b>	External Interrupt 0
<b>INT1</b>	External Interrupt 1
<b>INT2</b>	External Interrupt 2
<b>INT3</b>	External Interrupt 3
<b>INT4</b>	External Interrupt 4
<b>INT5</b>	External Interrupt 5
<b>INT6</b>	External Interrupt 6
<b>INT7</b>	External Interrupt 7
<b>RXRDY</b>	uart1 rx complete
<b>UDRE</b>	uart1 data register empty
<b>TXRDY</b>	uart1 tx complete
<b>RXRDY2</b>	uart2 rx complete
<b>UDRE2</b>	uart2 data register empty
<b>TXRDY2</b>	uart2 tx complete

The declaration of an interrupt generates automatically corresponding entries in the interrupt vector table, and also as a special code-frame for the interrupt procedure.

Here are at default all registers saved -> **complete register saving**.

This can be controlled by the compiler switches `{$NOSAVE}`, `{$NOREGSAVE}` and `{$NOSHADOW}`.

The global-interrupt-enable-flag, which is reset automatically by the CPU, is not changed, i.e. the interrupts stay disabled in runtime. If the 'end'- statement is reached, a RETI (return from interrupt) will be done, which enables the interrupt.

That means, that an interrupt procedure should be **as short as possible**, so the other interrupts are not disabled too long.

A **complete register saving** needs about 20 Bytes in RAM, statically, not on Stack or Frame. Because of this a stacked interrupt is impossible. That means within an interrupt procedure the interrupt never should be enabled again. The CPU itself does this enabling with the *RET* instruction

If an interrupt nesting is indispensable, the compiler switch *{\$NOSAVE}* must be used. So only the flags and the main work registers are saved. The procedures *PushAllRegs* and *PopAllRegs* can be used to save the remaining registers via the stack if necessary.

**Interrupt nesting is very dangerous and ends frequently in a system crash. It should be avoided whenever possible. At least a very careful planning an absolute necessity.**

### **Attention:**

the correct initialization of interrupts needs usually settings in different control and mask registers. With the contained drivers that are running in interrupt mode this is done by the corresponding drivers. If the application defines the interrupts, the compiler can not be of any assistance.

### **In these cases it is the duty of the application to do the necessary initializations !!**

If the system reaction is too time-critical and more statements need to be executed, the following procedure is recommended:

the interrupt increments a semaphore and returns. A process is always waiting for this semaphore and will get the control briefly.

```
Interrupt Int0;  
begin  
  IncSema (sema0);  
end;
```

```
Process ProcessInt0 (32, 16 : iData);  
begin  
  WaitSema (sema0);      {wait for sema0 > 0 }  
  ...  
  ...  
end;
```

Interrupt Service Routines can have own local variables. Because there is no Frame here, these vars are static/nonvolatile which means they are placed into the iData (SRAM) area and they are always accessible (static) and their content is always valid (nonvolatile).

Inside of the Interrupt procedure they can be accessed and used like any other var. Because of their static nature they can also be used by other parts of the application. But they are a property of a procedure so an access must be qualified:

```
Interrupt Timer1;  
var abc : byte;  
begin  
  abc:= 123;           // no qualification  
  ...  
end;  
...  
Interrupt_Timer1.abc:= $67; // qualify with "Interrupt_name"
```

Please note that "Interrupt\_" must always precede the qualifier.



# AVRco Compiler-Manual

Basically all Interrupts which are not supported by an interrupt Handler are handled by a default/dummy interrupt service routine with a simple "RETI". For debug and test purposes this interrupt error can be supported by the application. Normally such a condition should never occur so this routine should never be called by any interrupts. The implementation must be done in the application:

```
Interrupt IntErrorHandler;  
begin  
...  
end;
```

## 4.8.1.1 Push, Pop

```
Procedure Push (regnum : byte | regname : internal); //e.g. Push (24);  
Procedure Pop (regnum : byte | regname : internal); //e.g. Pop (_ACCFLO);
```

For a better readability. Creates the same code as  
ASM: PUSH ... ; or  
ASM: POP ... ;

## 4.8.1.2 PushAllRegs, PopAllRegs

Sometimes, but only sometimes, it makes sense to re-enable globals interrupts in an interrupt procedure for example a Timer Interrupt. This avoids too long interrupt disable times. In this case, if registers must be saved, this can not be done by the common automatic register save (Switch \$NOSHADOW inactive). This register save does not support "nested interrupts".

In order to save all registers in such a function where the global interrupt must be re-enabled, the register save must be done in a special way:

```
{$NoSave}  
Interrupt TIMER1COMPA; // TickTimer  
begin  
  PushAllRegs;  
  EnableInts;  
  ...  
  ...  
  PopAllRegs;  
end;
```

The switch {*\$NoSave*} is mandatory here.

This method should only be used if the global interrupts must be enabled in an interrupt service routine. The user should exactly know why and what he is doing here :-)

## 4.8.1.3 External Interrupts

The AVR family provides two external Interrupt types which can be controlled by port pins.

The first group contains these Pins/Interrupts where every pin can fire an own vector interrupt (unique vector). These are the interrupts INT0..INT7 and the associated port pins.

The second group contains the so called **PinChangeInterrupts** PCINT0..PCINTxx. Here always upto 8 pins of a port are joined together and firing one common interrupt vector.

## 4.8.1.4 Interrupt Pins INT0..INTx

These standard external interrupts are supported by the AVRco system in a way that if a concerned interrupt procedure is defined in the application, for example **Interrupt INT0**, the system enters the address of this procedure into the interrupt vector table. If this interrupt is raised the selected register saving is executed and this procedure is called. The initialisation of this interrupt in the associated Enable and Mask registers must be done by the application.

## 4.8.1.5 PinChangeInterrupts PCINT0..PCINT3 (\*4\*)

Most of the newer AVR CPUs provide this interrupt mechanism. Upto 4 Ports (PCINT0..3) or 32 interrupt pins are supported. This looks very powerful but it must be used with some restrictions. So there is only one vector for one port and each state change at every pin fires an interrupt. Means the Low/High edge fires and also the High/Low edge.

With many applications this is still sufficient. But sometimes it is necessary to identify the pin source of the interrupt. A separate vector or service routine per pin is necessary. So each PinChange should end in a specific service routine. The AVRco system supports this with a special interrupt handler **PCintServer** for the PCINT Interrupts.

Here the previous PIN state is compared with the actual one in order to find out which port pin was the reason for this interrupt. If then the application provides a special interrupt service routine so this one will be called and the register R16 (\_ACCB) contains a true or false (\$00/\$FF) dependend which edge of the pin change was the source of this interrupt. The associated interrupt procedurs must follow this naming conventions: PCINT00 upto PCINT31.

The Interrupt Handler *PCintServer* for the PCINT Interrupts must be imported.

```
From System Import PCINTserv0, PCINTserv1, ...;
```

In order to use the individual Pin Change Interrupts the necessary Interrupt procedures must be defined.

```
Interrupt PCint00; // PinB.0 mega168  
begin  
if _ACCB <> 0 then  
...  
else  
...  
endif;  
end;
```

PCINTxx is also supported by the Application Wizard and the Simulator.  
An example program can be found in the Demos Directory in [PCintServ](#).

## 4.8.2 TRAPS and Software Interrupts (SWI)

Using Software-Interrupts (SWI), also called Traps, is usually the only way with larger processors to switch from the application layer to the system layer (application level - system level). Many important operations are only enabled on the system level, e.g. IO-accesses, memory accesses into protected areas etc.

An additional reason for having SWI or Traps is the communication between one program part and another where neither part knows the structure, functions and addresses of the other part. Such types are Debug-Monitors for example.

Privileged layers don't exist with the AVR. Because of this there are no SWI or Traps. But in some cases it could be an advantage if they were available. It is possible to use hardware interrupt for these purposes, as Atmel suggests. But most of the interrupts of the AVR can't be used because a peripheral part or a pin interrupt must be wasted. Most of the mega CPUs provide a SPMRDY interrupt which is used very rarely, doesn't use any additional resources and is very simple to use.

The AVRco supports Traps through the SPMRDY. Not as elegant as native Traps or SWIs, but practical.



# AVRco Compiler-Manual

## Why Traps with AVR?

With the almost all AVR applications using Traps/SWI is definitely not necessary. There are only a very few special cases where they make sense. When a program is patched or parts of it do not “know” each other there can be the problem of communication between them. It is impossible to call any function of the other. And also the data transfer between them is very difficult.

A typical application is for example a Debug Monitor. Without Traps/SWI it must have its entry at a fixed known address. This is a necessity for the application to contact this monitor.

## How does it work?

The system provides the procedure **Trap (t : byte)** which can be called at anytime. The parameter “t” can be any byte which is passed to the receiving function. Assuming that the global interrupt is enabled the procedure “Trap” triggers a SPMRDY interrupt.

Either this interrupt then is processed by the interrupt procedure **Interrupt SPMRDY** or by a TrapHandler, which the application must then provide **TrapHandler (t : byte)**.

All this happens in the completely in the application. But how can the external part be involved? In the first case with “Interrupt SPMRDY” an entry address can be passed with the definition of the Trap mode which then will be called instead of the interrupt procedure in the application itself. So the app-internal interrupt procedure can be omitted. The same is true for the TrapHandler procedure.

In both cases if an external part is involved and Traps are used for communication this external entry address must be defined. This external address must be the absolute address of the external interrupt procedure or TrapHandler. This is similar as with most of the bigger architectures (16/32bit). This address is a **Byte**-address and must have an even value.

The parameter “t” is passed to the interrupt procedure “SPMRDY” in `_ACCA = R17`.

A special case are traps in the boot area. Here are interrupts not the right way. They can be used with another implementation that has nothing in common with this one. For more details see the chapter **BootTraps**.

### 4.8.2.1 Implementation of the Traps

#### Defines

The definition of the TrapHandler controls how the Traps are handled by the system.

```
Define ProcClock    = 8000000;      {Hertz}
          SysTick     = 10;          {msec}
          StackSize   = $0030, iData;
          FrameSize   = $0030, iData;
          TrapHandler = false, 0;    {Intproc only, no ext address}
```

With the define of the TrapHandler the first parameter (true/false) defines whether an interrupt or TrapHandler procedure is expected.

The second parameter DestAddr (longword) defines whether this procedure is a part of the application (0) or of the external part (> 0).

## **TrapHandler = false**

The *Trap(nn)* call triggers a SPMRDY interrupt. This interrupt then is executed either

- application-internaly (DestAddr = 0) or
- externally (DestAddr > 0).

The Interrupt procedure must be defined and handled in the same way as all other Interrupt procedures. The same restrictions are true like: short processing time, possible register save etc. The global Interrupt is still **disabled** at the entry time.

```
Interrupt SPMRDY;  
begin  
//_ACCA/R17 contains the parameter "t"  
end;
```

## **TrapHandler = true**

The *Trap(nn)* call triggers a SPMRDY interrupt. The TrapHandler then is executed either

- application-internaly (DestAddr = 0) or
- externally (DestAddr > 0).

The interrupt itself is completely handled by the system and calls the internal/external Handler procedure. At the entry time into the Handler the global interrupt is already **enabled**. So the Handler is a procedure without any restrictions.

```
Procedure TrapHandler(t : byte);  
begin  
...  
end;
```

## **DestAddr**

As described above this address is the entry address of the external Interrupt procedure or TrapHandler and must be an even byte address. So the Flash-end of the mega128 has the address \$1FFFE.

## **Note:**

If the DestAddr = 0 then the corresponding procedure (Interrupt or TrapHandler) must be defined in the application.

If the DestAddr > 0 then the corresponding procedure (Interrupt or TrapHandler) must be present externally. Declarations in the application itself are ignored by the system.

## **4.8.3 EXCEPTIONS**

With complex applications with many sub-routines and drives often there is a circumstance that in a deep level in a basic function (e.g. In/Out) there is an error or a time-out. Now this function can return a FALSE to the calling function, this function again returns also a false etc. until the call returns to this location where this action was initiated. Here the result (FALSE) can or must be interpreted which had its reasons "very deep down". This can work only when each concerned function returns a boolean and the calling function interprets this

```
if not funcx then  
Return(false);  
else ...
```

in a proper way.



# AVRco Compiler-Manual

The whole operation can be difficult and erroneous. For this purpose Borland introduced some years ago the elegant and powerful **Exceptions** in Delphi. Exceptions are error handlers. This means, if such an exception occurs the program immediately returns to this location where the exception handler was implemented. The difficult stepping up through all the functions used is discarded.

Exceptions always consist of an implementation part, limited by **Try** and **EndTry** and one or more **RaiseException** statements. Between **Try** and **EndTry** there can be an optional **Except** statement

```
Try  
StatementE..  
StatementE..  
StatementE..  
EndTry;
```

or

```
Try  
StatementE..  
StatementE..  
StatementE..  
except  
StatementN..  
StatementN..  
EndTry;
```

If the execution of a *Statement* meets the statement

```
RaiseException (num);
```

all further operations are discarded and the application returns to the initiating Try/EndTry block. In the first example the execution continues with the statement after the line "EndTry". In the second example the next executed statement is this one which follows the line "except". With the function

```
GetExceptResult
```

the parameter "num", which was passed with the function *RaiseException* can be read.

If there was no exception then with the first example all statements are executed. In the second example only the statements between *Try* and *Except* are executed and the execution jumps to the line after the *EndTry* statement.

The better way is always example two where the application always gets notified that an execution happened (raised) because the block between *Except* and *EndTry* is executed.

## **Attention**

An exception can only be raised (by *RaiseException(num)*) if it happens between the execution of *Try* and *EndTry* or *Except*. Otherwise it will be ignored. A *Try/EndTry* block always must be placed into a block. This is illegal:

```
if a > b then  
Try  
else  
Endtry;  
endif;
```



## Restriction

With MultiTasking there are some restrictions at this time. *Try/EndTry* must only be placed in the *MAIN*. So the *RaiseException* function is only valid if at this time the *MAIN* process is controlling the system. Otherwise it is ignored.

### 4.8.3.1 Implementation

#### Imports

The exception support must be imported.

*Import SysTick, TickTimer, ...;*

*From System Import Exceptions, ...;*

#### Defines

The Exceptions can be nested indirectly. The max. nesting must be defined

*Define Exceptions = 2[, Boot[, iData1]]; // 1..15 levels, Boot and iData1 are optional*

Exceptions must not be directly nested, but it is possible while a *Try/EndTry* is active additional ones can be placed into function statements. The count of the nested levels is limited by the above define. Each level needs a set of parameters which is placed into the RAM in a stacked order. If all *Try/EndTry* levels are occupied at runtime additional ones are ignored. The structure is built at runtime and released with an *EndTry*. So this can be called an Exception Stack.

If the Option *Boot* is selected then the Exception Handler is placed into the Boot area. Because this handler builds and uses some system variables the addresses of these vars must never be changed after a Flash download. Because this can not always be ensured by the application it makes sense that these vars will be placed into the already defined *Idata1* area (must be separately defined) where they are always be placed into the very first locations.

### 4.8.3.2 Functions

With the import of Exceptions two support functions also become imported:

*Procedure RaiseException (num : byte);*

This procedure raises the exception. Then the program immediately returns to the location which is on the first place (Top Level) of the exception stack. The parameter *num* is stored and can later be read back with the function described below. Please note that the global interrupt stays disabled from here on.

If the exception stack is empty, meaning there is no responsible *Try/EndTry* active at this time the *RaiseException* procedure will be ignored.

*Function GetExceptResult : byte;*

The parameter *num* passed by the function *RaiseException* can be read back by this function. The parameter can be used for several informations, e.g. It shows which Exceptions was the initiator or what kind of problem occurred.

#### Example Program:

An example can be found in the directory **..E-Lab\AVRco\Demos\Exceptions**



## 4.9 Statements

The statement part defines the action to be carried out by the program (or subprogram) as a sequence of *statements*; each specifying one part of the action. In this sense Pascal is a sequential programming language: statements are normally executed sequentially in time, not simultaneously. The statement part is enclosed by the reserved words **begin** and **end** and within it, statements are separated by semi-colons. Statements may be either *simple* or *structured*.

### 4.9.1 Simple Statements

Simple statements are statements which contain no other statements. These are the assignment statement, procedure statement, goto statement, and empty statement.

### 4.9.2 Assignment Statement

The most fundamental of all statements is the assignment statement. It is used to specify that a certain value is to be assigned to a certain variable. An assignment consists of a variable identifier followed by the assignment operator **:=** followed by an expression.

Assignment is possible to variables of any type as long as the variable (or the function) and the expression are of the same type or assignment compatible.

```
Angle:= Angle * 3;  
AccessOK:= False;  
AccessOK:= Answer = PassWord;  
Result:= (Entry * 13) shl 8;
```

### 4.9.3 Procedure Statement

A procedure statement serves to activate a previously defined user defined procedure or a pre-defined standard procedure. The statement consists of a procedure identifier, optionally followed by a parameter list, which is a list of variables or expressions separated by commas and enclosed in parentheses. When the procedure statement is encountered during program execution, control is transferred to the named procedure, and the value (or the address) of possible parameters are also transferred to the procedure. When the procedure finishes, program execution continues from the statement following the procedure statement.

```
Find (Name,Address);  
Sort (Address);  
Uppercase (Text);
```

### 4.9.4 Empty Statement

An 'empty' statement is a statement which consists of no symbols, and which has no effect. It may occur whenever the syntax of Pascal requires a statement but no action is to take place

```
repeat until KeyPressed; {wait for any key to be hit}
```

## 4.9.5 Structured Statement

Structured statements are constructs composed of other statements which are to be executed in sequence (compound statements), conditionally (conditional statements), or repeatedly (repetitive statements).

## 4.9.6 Compound Statement

A compound statement is used if more than one statement is to be executed in a situation where the Pascal syntax allows only one statement to be specified. It consists of any number of statements separated by semicolons. The block starts with a preliminary statement (e.g. *if .. then*) and ends with an associated reserved word (e.g. *endif*).

```
if Small > Big then  
  Tmp := Small;  
  Small := Big;  
  Big := Tmp;  
endif;
```

## 4.9.7 NOP Statement

The Pascal Statement "NOP" is implemented. It produces an Assembler "NOP"

```
Incl (bit);  
NOP;  
Excl (bit);
```

## 4.9.8 Conditional Statements

A conditional statement selects for execution a single one of its component statements.

In most cases decisions and branches are build with "*if..then..else*". This is a common and always usable way, but sometimes it is inefficient and code and time consuming.

If constants are used with decisions it is better to use this: "*if x in[a, b, m..p]*".

Another way is to use "case" constructs.

But in many cases there are no constants but variables and so "case" is unusable. But also here using powerful system functions like *IncToLim*, *DecToLim*, *IncToLimWrap*, *DecToLimWrap*, *ValueTrimLimit*, *ValueInTolerance*, *Lower*, *Higher* etc. can reduce code size in IF-constructs and also in general.

### 4.9.8.1 IF Statement

**IF**

The IF statement requires at least THEN and ENDIF

**THEN**

```
if a > b then a:= b; endif;
```

**ELSE**

```
if a > b then a:= b; else b:= a; endif;
```

**ELSIF**

```
if a > b then ...; elsif b = a then ...; endif;
```

**ENDIF**

End of an IF Statements

*IF* is the leading statement of a branch. After *IF* **has to** be an operation with a boolean result (*true/false*). This can be a compare ( $a > b$ ) or abstractly a boolean (*bit* or *var*).

After the operation there **has to** be a *then*. The *begin*, which is normal in Pascal, is not permitted in this case (see also description of *begin*). After *then* there has to be at least 1 executable statement. Then optional an *else* or *elsif* can follow, whereby *begin* is not permitted, too, and there has to be at least one executable statement.

In contrast to Standard-Pascal *IF* has always to be terminated with *ENDIF*.

In certain cases it makes sense to work with *GoTo* instead of *If*, but this should be an exception.

### **Attention:**

Nested *IF*'s are permitted, but they have to be programmed very carefully, because provisional results are generally stored on the stack. In case of doubt (Stack-Overflow) the corresponding program code has to be tested with help of a debugger/simulator. Sometimes using *elsif* is better than a nested *if*.

**if**  $a > b$  **then**

...

**elsif**  $a = b$  **then**

...

**else**            {*a* is less than *b*}

...

**endif;**

### 4.9.8.2 GOTO Statement

*GOTO* is an absolute branch statement, which leads to any location (forwards and backwards) in a **block** (block definition see below), and that is the danger. The branch destination always is a label, which has to be declared beforehand. Within a block only **one label definition** and so only **one branch destination** is permitted. But **Several GOTOs** can point to one destination.

A *goto* statement consists of the reserved word **goto** followed by a label identifier. It serves to transfer further processing to that point in the program text which is marked by the label.

The following rules should be observed when using *goto* statements:

- 1) Before use, labels must be declared. The declaration takes place in a label declaration in the declaration part of the block in which the label is used.
- 2) The scope of a label is the block in which it is declared. It is thus not possible to jump into or out of procedures and functions.

*Procedure GotoSample;*

**begin**

**Label:** *lab1;*            {*Definition of a Labels*}

...

**Goto** *Lab1;*

...

**Lab1:**                    {*Branch destination, no Semicolon!!*}

...

**IF**  $a > b$  **then**

**Label:** *Lab2;*

**Lab2:**                    {*Branch destination, no Semicolon!!*}

...

**goto** *Lab2;*

...

**endif;**

...

**goto** *Lab1;*

**end;**

## Block Definition

In Pascal blocks are determined by certain block limits. A typical block limit is for example the BEGIN and END of a procedure or function. Loop blocks are limited by LOOP..ENDLOOP, REPEAT..UNTIL, WHILE..ENDWHILE, FOR..ENDFOR. The statements between IF..ENDIF, IF.. ELSE, IF..ELSIF, ELSIF..ELSIF, ELSIF..ELSE, ELSE..END also form **respectively one block**.

All three statements (label, Goto and the destination) of a Goto have to be within such a block. But it is irrelevant if the block is split in sub-blocks (see *Begin IF..ENDIF END*). In above mentioned example LAB2 is only recognized within IF..ENDIF, while Lab1 is known in the whole procedure, however not in the subblock IF..ENDIF.

## Comment:

in C block limits are determined with { and } and so very clear :-)

## **GOTO**

The Goto statement requires a definition of the label and use of this label in the same block.

## **LABEL**

The label defines the destination address of the Goto statement.

The programmer always should take care to produce a readable and hence easily maintainable code. But this described GoTo-statement generally contradicts this aim. Sometimes, and only sometimes, a GoTo helps to get a better view over the program. Because of that GoTo should only be used in emergency, because in most cases an If- or loop-statement is much more elegant and safe.

### **4.9.8.3 CASE Statement**

The case statement consists of an expression (the selector) and a list of statements, each preceded by a case label of the same type as the selector. It specifies that the one statement be executed whose case label is equal to the current value of the selector. If none of the case labels contain the value of the selector, then either no statement is executed, or, optionally, the statements following the reserved word **else** are executed. The else clause is an expansion of standard Pascal.

A case label consists of one to many constants or subranges separated by commas followed by a colon. A subrange is written as two constants separated by the subrange delimiter '..'. The type of the constants must be the same as the type of the selector. The statements following the case label are executed if the value of the selector equals one of the constants or if it lies within one of the subranges.

The case label may be any 8 bit data type such as *byte*, *Int8*, *enum* or *char* and also *word* or *integer* (16 bit).

Case is the leading statement of a branch block. After case an 8bit variable **has to** follow (byte, char or enum) and then the word "of" (case v1 of ..). Within this block there are instructions what to do, if a variable is identical with a constant or a constant area (7 :... or 8..12 :...).

This area-declaration has to be terminated with a colon. The following area-block must contain at least one statement. Every single statement **has to** be terminated with a ';'. This sub-block is to abort with a separator-symbol ( | ).

After one or several area-blocks there can be an optional **Else** with further statements. The EndCase-statement generally forms the termination. Case is actually a specialized If-statement.

```
const c1 = 83;  
        c2 = 105;
```

```
Case x of  
  0      : inc(a);  
        |  
  1, 7   : a:= 4;  
        |  
  2..6   : x:= x + a;  
        | dec(x);  
        |  
  8..9, 12 : PWMport1 (45);  
        |  
  14..23,  
  27..67  : a:= a * a;  
        |  
  c1..c2  : a:= c1;  
        |  
else  
  x:= 0;  
EndCase;
```

## CASE

Begin of a Case Block.

**Case** x **of** ...

## ENDCASE

End of a Case Block.

## 4.9.8.4 FOR Statement

For is the leading statement of a program-loop. After For, an 8 or 16bit runtime-variable (control variable) has to follow. A start value has to be assigned to this variable by := . Then follows the operator **TO** or **DownTo**, which specifies whether the runtime-variable should be incremented or decremented within every pass. The optional argument **BY** determines the increment or the decrement-value (1..255). Then follows the termination-value and the instruction **DO**. The start-value and the termination value may be 8 or 16bit variable or constants.

After the head-declaration (For x:= 0 to v1 do ) follows a block with instructions, which is to be executed on each pass. This block can be empty.

The number of passes is depending on the difference between termination- and start-value.

With **for i:= 2 to 1** the loop is not executed since **i** initially has the value **2**

With **for i:= 2 to 2** the loop is executed once and **i** then has the value **3**

With **for i:= 2 to 7** the loop is executed six times and **i** then has the value **8**

In contrast to Standard-Pascal **FOR** has to be terminated with **ENDFOR**.

```
const a = 0;  
var v1 : byte;  
    x  : byte;
```

```
for x:= a to v1 do           {ramp up}  
    PWMport1:= x;  
endfor;
```

```
for x:= x downto 0 do       {ramp down}  
    PWMport1:= x;  
endfor;
```

A FOR-loop can be terminated with a break-statement:

```
for x:= 1 to 9 by 2 do  
    ...  
    if a:= 0 then Break;  
    ...  
endfor;
```

See also **Continue!**

### FOR

Begin of a For loop

```
for a:= 0 to 45 do inc (x); endfor;
```

### ENDFOR

End of a For loop

## 4.9.8.5 WHILE Statement

While is the leading statement of a program-loop. After While an operation with a boolean result (true/false) **must** follow. This can be a compare ( $a > b$ ) or a boolean (bit or var). The expression controlling the repetition must be of type Boolean. The statements contained in the loop are repeatedly executed as long as *expression* is *True*. If its value is *false* at the beginning, the statement is not executed at all. The loop is repeated as long as the controlling expression returns *true*.

The statement **must** be terminated with a **do**. The *begin*, which is usual in Pascal, is not permitted in this case (see also description of *begin*). After *do* there should be at least one executable statement. Leaving the while-loop is possible with a **BREAK**. See also **Continue!**

In contrast to standard-Pascal *WHILE* always has to be terminated with *ENDWHILE*.

```
while  $x < 100$  do  
  inc ( $x$ );  
endwhile;
```

### WHILE

Begin of a While loop

```
while  $a < b$  do inc ( $a$ ); endwhile;
```

### ENDWHILE

End of a While loop

## 4.9.8.6 REPEAT Statement

REPEAT is the leading statement of a program-loop. After REPEAT there **can** be one or several executable statements. The *begin*, which is usual in Pascal, is not permitted in this case (see also description of *begin*).

The statement UNTIL forms the termination of the loop. After that there has to follow an operation with a boolean result (true/false). This can be a compare ( $a > b$ ) or a boolean (bit or var). The loop is executed as long as the operation returns false. So in contrast to *WHILE* it is executed at least once.

Leaving the loop is possible with **BREAK**. See also **Continue!**

```
 $x := 0$ ;  
repeat  
  inc( $TCC$ );  
until  $TCC > 20$ ;
```

### REPEAT

Begin of a Repeat loop

### UNTIL

End of a Repeat loop

## 4.9.8.7 CONTINUE

The Continue statement interrupts the normal flow of program execution of a **for-**, **while-** or **repeat-**loop and transfers execution to the beginning of the loop. The following statements until EndFor, EndWhile or Until are ignored.



## 4.9.8.8 LOOP Statement

Within a Controller-application the program normally never terminates, i.e. the program runs in an endless-loop. In this case Pascal only offers the construction: *Repeat ... until false = true;*  
This is formally o.k., but is nonsense.

Modula-2 allows the endless-loop *LOOP* and this is implemented, too.  
All statements within *Loop ... Endloop;* are repeatedly processed.

*Loop .. Endloop;* is also permitted within a procedure or even within a loop.  
The statement *ExitLoop* is necessary to exit.

**ExitLoop** exits the loop and continues with the next statement after *EndLoop*.

```
begin  
  loop  
    ...  
  endloop;  
end.
```

### LOOP

Begin of an endless-loop

### ENDLOOP

End of an endless-loop

### EXITLOOP

Termination of an endless loop

```
loop  
  ...  
  ...  
  if a > b then  
    exitloop;  
  endif;  
  ...  
endloop;
```



## 4.10 System Library - Standard

### 4.10.1 TRUE

Pre-defined constant = \$FF

### 4.10.2 FALSE

Pre-defined constant = \$00

### 4.10.3 PI

Pre-defined constant = 3.141592654 (float)

### 4.10.4 NIL

Pre-defined constant of type pointer = \$0000

*p := nil;*

## 4.10.5 Type Conversion

### 4.10.5.1 BOOLEAN

Converts the argument into a boolean

*Function Boolean (a : type) : boolean;*

*bo := boolean (x);*

### 4.10.5.2 BYTE

Converts the argument into a byte

*Function Byte (a : type) : byte;*

*b := byte (x);*

### 4.10.5.3 Int8

Converts the argument into a Int8

*Function In8(a : type) : int8;*

*i := Int8 (x);*

### 4.10.5.4 CHAR

Converts the argument into a char

*Function Char (a : type) : char;*

*ch := char (x);*

## 4.10.5.5 WORD

Converts the argument into a word

**Function** *Word* (*a : type*) : *word*;

*w:= word* (*x*);

## 4.10.5.6 INTEGER

Converts the argument into an integer

**Function** *Integer* (*a : type*) : *integer*;

*i:= Integer* (*x*);

## 4.10.5.7 LONGWORD

Converts the argument into a LongWord

**Function** *LongWord* (*a : type*) : *longword*;

*ww:=longword* (*x*);

## 4.10.5.8 LONGINT

Converts the argument into LongInt

**Function** *LongInt* (*a : type*) : *longint*;

*Li:= LongInt* (*x*);

## 4.10.5.9 FLOAT

Converts the argument into a float

**Function** *Float* (*a : type*) : *float*;

*f:= Float* (*x*);

## 4.10.5.10 FLOATASLONG

Converts the argument into a LongWord. But there is no converting or processing at all.

**Function** *FloatAsLong* (*f : float*) : *longword*;

**structconst**

*lws : longword = FloatAsLong* (1.234);

**const**

*lwc : longword = FloatAsLong* (1.234);

// var

*lw:= FloatAsLong* (*fc*);

*lw:= FloatAsLong* (*fs*);

## 4.10.5.11 LONGASFLOAT

Converts the argument into a Float. But there is no converting or processing at all.

**Function** *LongAsFloat* (*L : LongWord*) : *float*;

### **strutconst**

*fs : Float = LongAsFloat (\$12345678);*

### **const**

*fc : Float = LongAsFloat (\$12345678);*

*// var*

*f:= LongAsFloat (lws);*

*f:= LongAsFloat (lwc);*

## 4.10.5.12 POINTER

Converts the argument into a pointer

**Function** *Pointer* (*a : type*) : *pointer*;

*p:= Pointer (x);*

## 4.10.6 Character and String Functions

### 4.10.6.1 ORD

Ordinal number of a symbol/character.

**Function** *Ord* (*ch : char*) : *byte*;

*b:= ord ('a');            { \$61 }*

### 4.10.6.2 UPCASE

Converts char into capitals.

**Function** *Uppcase* (*ch : char*) : *char*;

*ch:= UpCase (ch);*

### 4.10.6.3 LOWCASE

Converts char into lowercase letters.

**Function** *LowCase* (*ch : char*) : *char*;

*ch:= LowCase (ch);*

### 4.10.6.4 UPPERCASE

Converts String into capitals.

**Function** *Uppercase* (*st : string*) : *string*;

*st:= UpperCase (st);*

## 4.10.6.5 LOWERCASE

Converts String into lowercase letters.

**Function** *LowerCase* (*st* : string) : string;

*st* := *LowerCase* (*st*);

## 4.10.6.6 COPY

Returns a string which is a substring from the source, beginning at “pos“ and has the length “count“. The destination must be a string var. Can be used in conjunction with concatenate.

**Function** *Copy* (*st* : string; *pos*, *count* : byte) : string;

*st* := *Copy* (*st*, 2, 3);

## 4.10.6.7 STRREPLACE

The string *src* overwrites the destination string *dest* at position *pos*. The length of the destination string remains as previous.

**Procedure** *StrReplace* (*src* : string; **var** *dest* : string; *pos* : byte);

## 4.10.6.8 TRIM

Removes leading and trailing spaces from the string.

**Function** *Trim* (**const** *st* : string) : string;

## 4.10.6.9 TRIMLEFT

Removes leading spaces from the string.

**Function** *TrimLeft* (**const** *st* : string) : string;

## 4.10.6.10 TRIMRIGHT

Removes trailing spaces from the string.

**Function** *TrimRight* (**const** *st* : string) : string;

## 4.10.6.11 PADLEFT

Inserts leading spaces (or the optional Pad Char) into the string. The string becomes extended so that the new length is now **len**. If the origin length  $\geq$  len then nothing changes.

**Function** *PadLeft* (**const** *st* : string; *len* : byte [*;pad* : char]) : string;

## 4.10.6.12 PADRIGHT

Appends spaces (or the optional Pad Char) to the string. The string becomes extended so that the new length is now **len**. If the origin length  $\geq$  len then nothing changes.

**Function** *PadRight* (**const** *st* : string; *len* : byte [*;pad* : char]) : string;



## 4.10.6.13 LENGTH

Function returns the actual length of a string.

**Function** *Length* (*s* : string) : byte;

*x* := *length* (*st1*);

## 4.10.6.14 SETLENGTH

This procedure changes the length of a string to the desired length, in the limits of the underlying string type.

**Procedure** *SetLength* (*st* : string; *len* : byte);

*SetLength* (*st*, 6);

## 4.10.6.15 POS

Function returns the position of a character within a string.

**Function** *Pos* (*a* : char; *s* : string) : byte;

*x* := *Pos* (*ch1*, *st1*);

## 4.10.6.16 POSN

Function returns the position of a character within a string. Search starts at a given position.

**Function** *PosN* (*a* : char; *s* : string; *start* : byte) : byte;

This is a non-standard Pascal function. It is functionally similar to the "POS" function in that it can be used to find a char in a string. POS searches for the char starting at the beginning of the string to search. It can find only one occurrence of the char. POSN on the other hand allows you to start the search from any position inside the string, thus POSN can be used to find multiple occurrences of the char.

POSN returns the position of "char" in "string". The first possible position in string is "1". The result is of type byte. If "char" cannot be found, the function returns a "0". "position" may be any number from 1 to 255 subject to the actual string length. A search past the end of the string will return a "0" result. POSN(char, string, 1) is the equivalent of POS(char, string).

"string" must refer to an existing string variable. It may not be a string expression.

**var**

*S* : string[30];

*N* : byte;

**begin**

*S* := 'Hello World Hello World';

*N* := *PosN* ('W', *s*, 10);                    //N will be set to 19

**end;**

## 4.10.6.17 APPEND

Procedure, appends the string "src" to the end of the string "dst" (concat).

*Procedure Append (src : string; var dst : string);*

The maximum length of "dst" will be not exceeded.

## 4.10.6.18 INSERT

Procedure, inserts a string into a string.

*Procedure Insert (src : string; var dst : string; p : byte);*

*Insert ('abc', st1, 4);*

## 4.10.6.19 DELETE

Procedure, deletes a number of characters within a string.

*Procedure Delete (var s : string; pos, count : byte);*

*Delete (st1, 3, 2);*

## 4.10.6.20 STRCLEAN

Function, removes/replaces control characters from a string.

*Function StrClean (const st : string; gt127 : boolean; subst : char) : string;*

The string "st" is scanned for chars < \$20 (space) and if the boolean "gt127" is true, also chars > \$7F (> #127) are scanned for. If such a char is found it will be replaced by the char "subst" in case subst is > #0. If subst = #0 then char found will be deleted and not replaced.

## 4.10.6.21 STRTOINT

Function, converts a string to a Byte, Word, Int8, Integer, LongWord, LongInt, Word64 or Int64 depending on the destination variable. The function accepts variables in RAM or EEPROM. Strings in flash are not supported.

*Function StrToInt (st : string) : byte; {integer ..}*

*w:= StrToInt (st);*

string formats:

decimal string:	"1234"
hexadecimal string:	"\$ABCD"



# AVRco Compiler-Manual

## 4.10.6.22 HEXTOINT

With *StrToInt* the hexadecimal string has to begin with a dollar sign "\$".  
But there are already fixed strings, for example out of a communication, which receive without a "\$".  
So it makes sense to convert these strings without the insertion of a "\$" at string[1]:

**Function** *HexToInt* (*st* : string) : integer [byte, int8, word, longint, longword];

expects a hex-string but without a leading "\$"

## 4.10.6.23 STRTOFLOAT

Converting of a string into a float value. The decimal point is defined by the global constant "*DecimalSep*" default ".", which can be redefined

**Function** *StrToFloat* (*st* : string) : float;

*f* := *StrToFloat* (*st*);

## 4.10.6.24 STRTOARR

Generation of Null-terminated strings (C Strings).

**Function** *StrToArr* (*var st* : string) : array of char

copies characters out of the string into the array until either the strings end is reached or the array is full.  
In both cases the last transferred character is always a null. This function is only applicable in an assignment.

*array* := *StrToArr* (*st*);

The string as the source must reside either in RAM, Flash or EEPROM. Functions as a source are not allowed.

## 4.10.6.25 ARRTOSTR

Processing of Null-terminated Strings (C Strings).

**Function** *ArrToStr* (*arr* : array of char) : string;

copies characters out of the array into the string until either a null is found or the end of the array is reached or the string is full.

The null is not transferred. This function is only valid in an assignment statement:

*string* := *ArrToStr* (*ar*);

The string as the source must reside either in RAM, Flash or EEPROM. Functions as a source are not allowed.

## 4.10.6.26 EXTRACTFILEPATH

returns the path-part of a FileName if present.

"A:FileName.ext" -> "A:" "ppp\nnnn" -> "ppp\"

**Function** *ExtractFilePath* (*FName* : string) : string;



## 4.10.6.27 EXTRACTFILENAME

returns the name-part of a FileName  
"A:FName.ext" -> "FName" "ppp\nnnn" -> "nnnn"

**Function** *ExtractFileName (FName : string) : string;*

## 4.10.6.28 EXTRACTFILEEXT

returns the extension-part of a FileName if present.  
"A:FName.ext" -> "ext" "nnnn.ee" -> "ee"

**Function** *ExtractFileExt (FName : string) : string;*

## 4.10.7 Access to Parts of Variable / Constants

### 4.10.7.1 SWAP

Swaps LoByte with a HiByte of Word or Integer or the LoNibble with the HiNibble of a Byte respective a Char. For 32bit types the low-word is swapped with the high-word.

**Function** *Swap (x : type) : type;*

*X:= Swap (i);*

### 4.10.7.2 SWAPLONG

Sometimes it is necessary to make a mirrored value of a 32bit value.  
The first and the last byte exchange their position. The same is true for the two middle bytes.

**Function** *SwapLong (const x : LongWord|LongInt) : LongWord|LongInt;*

*longVal:= SwapLong (longVal);*

If the variable "longVal" had the value \$12345678 before the operation, so after the function call it contains the value \$78563412.

**Attention:** don't confuse with *Swap (longVal) !!*

### 4.10.7.3 MIRROR8

Mirrors the argument. Exchange bit7 <-> bit0, bit6 <-> bit1, ...

**Function** *Mirror8 (b : byte|int8|char) : byte|int8|char;*

### 4.10.7.4 MIRROR16

Mirrors the argument. Exchange bit15 <-> bit0, bit14 <-> bit1, ...

**Function** *Mirror16(w : word|integer) : word|integer;*

### 4.10.7.5 MIRROR32

Mirrors the argument. Exchange bit31 <-> bit0, bit30 <-> bit1, ...

**Function** *Mirror32(Lw : longword|longint) : longword|longint;*



## 4.10.7.6 LONIBBLE

returns the Low-nibble of a byte (lower 4bits)

**Function** *LoNibble* (*b* : *byte|Int8*) : *byte|int8*;

## 4.10.7.7 LO (Function)

returns least significant byte of a 16bit value

**Function** *Lo* (*w* : *word*) : *byte*;

**Function** *Lo* (*i* : *integer*) : *byte*;

*a* := *lo* (*i*);

## 4.10.7.8 LO (Assignment)

assignment to the Low byte of a word

*LO* (*word*) := *byte*;

## 4.10.7.9 LOWORD (Function)

returns the Low word of a 32bit value (LongInt or LongWord)

**Function** *LoWord* (*ww* : *Longword*) : *word [integer]*;

**Function** *LoWord* (*ii* : *LongInt*) : *integer [word]*;

*a* := *loWord* (*i*);

## 4.10.7.10 LOWORD (Assignment)

assignment to the Low Word of a LongWord

*LOWORD* (*long*) := *word*;

## 4.10.7.11 HINIBBLE

returns the High-nibble of a byte (higher 4bits)

**Function** *HiNibble* (*b* : *byte|int8*) : *byte|int8*;

## 4.10.7.12 HI (Function)

returns most significant byte of a 16bit value

**Function** *Hi* (*w* : *word*) : *byte*;

**Function** *Hi* (*i* : *integer*) : *byte*;

*a* := *hi* (*i*);

## 4.10.7.13 HI (Assignment)

assignment to the High byte of a word

*HI* (*word*) := *byte*;

## 4.10.7.14 HIWORD (Function)

returns the High word of a 32bit value (LongInt or LongWord)

**Function** *HiWord* (*ww* : Longword) : word [integer];

**Function** *HiWord* (*ii* : LongInt) : integer [word];

*a* := *hiWord* (*i*);

## 4.10.7.15 HIWORD (Assignment)

assignment to the High Word of a LongWord

*HIWORD* (*long*) := *word*;

## 4.10.8 ABS

Absolute value of an Int8, Integer, Longint or Float value

**Function** *Abs* (*i* : integer) : integer;

**Function** *Abs* (*f* : float) : float;

*a* := *abs* (*a*);

## 4.10.9 Negate

The negative value (Two's Complement) of a Byte, Int8..Longword, LongInt or Float value

**Function** *Negate* (*v* : type) : type;

*a* := *Negate* (*a*);

## 4.10.10 INC

Variables only increment for byte, int8, word, integer, longword and longint. Argument is wrapped if the limits are exceeded.

**Procedure** *Inc* (*var v* [, *step*] : type);

*inc* (*a*);

## 4.10.11 INCTOLIM

**Function** *IncToLim* (*var v* : ordinal [, *limit* : ordinal[; *val* : ordinal]]) : boolean;

The argument "v" is incremented, provided that "v" has not reached it's natural limit.

If the optional parameter "limit" is given, then this serves as the limit. If the function was successful, this means there was an increment, and the function returns true. Otherwise it returns false.

The optional parameter "val" specifies the increment value.

Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint

The function is extremely fast and short. It is very well suited for loop implementations.

*var i* : integer;

**repeat**

...

**until not** *IncToLim* (*i*, 1000);

The loop will be repeated until "i" becomes the value 1000.

## 4.10.12 INCTOLIMWRAP

**Function** *IncToLimWrap* (**var** *value*, *lim*, *pres* : *type*) : *boolean*;

Increments the variable "value" by 1. If this exceeds the value of "lim" the variable "value" is reset to the content of "pres" and the result returns true. Otherwise the result is false.

Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint

## 4.10.13 DEC

Variables only decrement for byte, Int8, word, integer, longword and longint. Argument is wrapped if the limits are exceeded.

**Procedure** *Dec* (**var** *v* [, *step*] : *type*);

*dec* (*b*);

## 4.10.14 DECTOLIM

**Function** *DecToLim* (**var** *v* : *ordinal* [, *limit* : *ordinal*; *val* : *ordinal*]) : *boolean*;

The argument "v" is decremented, provided that "v" has not reached it's natural limit.

If the optional parameter "limit" is given, then this serves as the limit. If the function was successful, this means there was a decrement, and the function returns true. Otherwise the function returns false.

The optional parameter "val" specifies the decrement value.

Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint

The functions is extremely fast and short. It is very well suited for loop implementations.

**var** *i* : *integer*;

**while** *DecToLim* (*i*) **do**

...

**endwhile**;

The loop will be repeated until "i" becomes the value -32768.

## 4.10.15 DECTOLIMWRAP

**Function** *DecToLimWrap* (**var** *value*, *lim*, *pres* : *type*) : *boolean*;

decrements the variable "value" by 1. If this exceeds the value of "lim" the variable "value" is reset to the content of "pres" and the result becomes true, otherwise false.

Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint

## 4.10.16 VALUETRIMLIMIT

**Function** *ValueTrimLimit* (*value*, *vmin*, *vmax* : *type*) : *type*;

compares the content of "value" to the two limits "vmin" and "vmax", where vmin must always be smaller than vmax. If one of the two limits is exceeded the result becomes the content of this limit, otherwise the origin content of value is returned.

Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint, Float

## 4.10.17 VALUEINTOLERANCE

**Function** *ValueInTolerance* (*value, ref, tol : type*) : *boolean*;

compares the content of "value" to the limit "vmin" which is build from (ref - tol) and "vmax", build from (ref + tol). If one of the two limits is exceeded the result returns false. Otherwise it returns true.  
Type = Byte, Int8, Word, Integer, Longword, Longint, Float

## 4.10.18 VALUEINTOLERANCEP

**Function** *ValueInToleranceP* (*value, ref : type; tol : byte*) : *boolean*;

compares the content of "value" with the limit "vmin" which is build from (ref - (ref div 100) \* tol) and "vmax", build from (ref + (ref div 100) \* tol). If one of the two limits is exceeded the result is false. Otherwise it is a true.

The value "tol" must be in the range of 0..100 because it is used as "percent". This function is the same as the function "*ValueInTolerance*" except that here the tolerance is not absolute but relative in percent.  
Type = Byte, Int8, Word, Integer, Longword, Longint, Float

## 4.10.19 VALUEINRANGE

**Function** *ValueInRange* (*value, vmin, vmax : type*) : *boolean*;

compares the variable "value" with the two limits "vmin" and "vmax" where vmin must be smaller than vmax. If value exceeds one of two limits the function returns False. Otherwise it returns true.  
Type = Enum, Byte, Int8, Char, Word, Integer, Longword, Longint, Float

## 4.10.20 MULDIVBYTE

With division of bytes the result is imprecise in many cases. Only with the help of a trick it's possible to multiply a byte by a non-integer value. The following is not possible, normally:

```
b:= b * 0.2;
```

But one can replace the above by:

```
b:= (b * 10) div 50;
```

This works in many cases, but fails if the result of the multiplication is greater as the bounds of a byte (0..255).

```
b:= (100 * 100) div 250;
```

The multiplication results in an byte-overflow and the total result is completely wrong. To avoid this problem, one can do the whole operation in word. But this means that the values used must be converted to word, which results in a larger and slower program.

**Function** *MulDivByte* (*a1, a2, d : byte*) : *byte*;

The function calculates the 16bit result of the multiplication and divides this result by the 8bit divisor. This assures that an overflow error is impossible. The result of this function must fit into a byte.

```
bb:= 100; // byte  
bb:= MulDivByte (ww, 99, 100); // -> bb:= bb * 0.99
```

## 4.10.21 MULDIVINT8

Similar function as MulDivByte or MulDivInt but for ShortInt = Int8.

**Function** *MulDivInt8* (*a1, a2, d : Int8*) : *Int8*;

## 4.10.22 MULDIVINT

With division of integers the result is imprecise in many cases. Only with the help of a trick it's possible to multiply an integer or Word by a non-integer value.

The following is not possible, normally:

```
i := i * 0.27;
```

But one can replace the above by:

```
i := (i * 100) div 370;
```

This works in many cases, but fails if the result of the multiplication is greater as the bounds of an integer (+/- 32767).

```
i := (1000 * 100) div 370;
```

The multiplication results in an integer-overflow and the total result is absolutely wrong. To avoid this problem, one can do the whole operation in LongInt. But this means that longints must be imported, which results in a larger and slower program.

**Function** *MulDivInt* (*a1, a2, d : integer*) : *integer*;

**Function** *MulDivInt* (*a1, a2, d : word*) : *word*;

The function calculates the 32bit result of the multiplication and divides this result by the 16bit divisor. This assures that an overflow error is impossible. The result of this function must fit into an integer or Word. The first parameter defines the principal operation. If it is an integer, a signed calculation is executed, otherwise it is an unsigned one.

```
ww := 1000; // word  
ww := MulDivInt (ww, 2, 3); // -> ww := ww * 0.666
```

```
ii := -1000; // integer  
ii := MulDivInt (ii, 100, 125); // -> ii := ii * 0.8
```

## 4.10.23 SQUAREDIVBYTE

**Function** *SquareDivByte* (*val, divfact : byte*) : *byte*;

Calculate the square of a value and divide the result by another value. The advantages of these functions are that an overflow of the square result does not matter

## 4.10.24 SQUAREDIVINT8

**Function** *SquareDivInt8* (*val, divfact : int8*) : *int8*;

Calculate the square of a value and divide the result by another value. The advantages of these functions are that an overflow of the square result does not matter

## 4.10.25 SQUAREDIVINT

**Function** *SquareDivInt* (*val*, *divfact* : *word|integer*) : *word|integer*;

Calculate the square of a value and divide the result by another value. The advantages of these functions are that an overflow of the square result does not matter

## 4.10.26 INTEGRATEB

**Function** *IntegrateB* (*oldVal*, *newVal*, *fact* : *byte*) : *byte*;

Integrate a new value with an existing.

Calculation: *result:= ((oldVal \* fact) + newVal) div (fact + 1)*;

An overflow in the multiplication cannot happen because the internal maths is done with the next bigger type.

## 4.10.27 INTEGRATEI8

**Function** *IntegrateI8* (*oldVal*, *newVal* : *int8*; *fact* : *byte*) : *int8*;

Integrate a new value with an existing.

Calculation: *result:= ((oldVal \* fact) + newVal) div (fact + 1)*;

An overflow in the multiplication cannot happen because the internal maths is done with the next bigger type.

## 4.10.28 INTEGRATEI

**Function** *IntegrateI* (*oldVal*, *newVal* : *integer*; *fact* : *byte*) : *integer*;

Integrate a new value with an existing.

Calculation: *result:= ((oldVal \* fact) + newVal) div (fact + 1)*;

An overflow in the multiplication cannot happen because the internal maths is done with the next bigger type.

## 4.10.29 INTEGRATEW

**Function** *IntegrateW* (*oldVal*, *newVal* : *word*; *fact* : *byte*) : *word*;

Integrate a new value with an existing.

Calculation: *result:= ((oldVal \* fact) + newVal) div (fact + 1)*;

An overflow in the multiplication cannot happen because the internal maths is done with the next bigger type.

## 4.10.30 Even

Tests the value for even. Function returns True if the argument is even.  
Only for byte, int8, word, integer, longword and longint

**Function** *Even*(*x* : *type*) : *boolean*;

*If Even(V1) then ..*

## 4.10.31 ODD

Tests the value for odd. Function returns True if the argument is odd.  
Only for byte, int8, word, integer, longword and longint

**Function** *Odd* (*x* : *type*) : *boolean*;

*If Odd (V1) then ..*



## 4.10.32 PARITY

Function, returns the parity, even/odd of a byte or char. True, if parity is odd

**Function** *Parity* (*const b : byte|char*) : *boolean*;

*Bool:= Parity (bb);*

## 4.10.33 ISPOWOFTWO

This function checks the number "n" whether is a power of 2. Valid parameters are Byte..LongInt.

**Function** *IsPowOfTwo* (*n : type*) : *boolean*;

## 4.10.34 SIGN

Function, returns the sign of a number as a boolean:

**Function** *Sign* (*const num : integer[int8, longint, float]*) : *boolean*;

If the argument is positive the function returns a true otherwise a false.

## 4.10.35 SGN

Function, returns the sign of a number as an integer:

**Function** *Sgn* (*const num : integer[int8, longint, float]*) : *integer*;

If the argument is > 0 the function returns a '1'. If the argument is zero the function returns a zero otherwise a '-1'.

## 4.10.36 PRED

The function returns the next-lower value of a variable. The type has to be ordinal, for example byte, word, integer. The limits of the type given are not exceeded. There is no "wrap". The predecessor of Byte 0 is always 0.

**Function** *Pred* (*x : type*) : *type*;

*x:= Pred (y);*

## 4.10.37 SUCC

The function returns the next-higher value of a variable. The type has to be ordinal, for example byte, word, integer. The limits of the type given are not exceeded. There is no "wrap". The successor of Byte 255 is always 255.

**Function** *Succ* (*x : type*) : *type*;

*x:= Succ (y);*



## 4.10.38 MIN

Returns the lowest possible value of the type.

**Function** *Min* (*x: type*) : *type*;

*x := min (a);*

## 4.10.39 MAX

Returns the highest possible value of the type.

**Function** *Max* (*x: type*) : *type*;

*x := max (a);*

## 4.10.40 SIZEOF

Function returns the required memory of an object in bytes.

**Function** *SizeOf* (*x : type*) : *word*;

*x := SizeOf (a);*  
*x := SizeOf (st1);*     *{whole memory area of a string}*

## 4.10.41 BitCountOf

Function returns the count of bits of an ordinal which have the value of "1".

**Function** *BitCountOf* (*x : ordinal*) : *byte*;

*n := BitCountOf (a);*

## 4.10.42 ADDR

*p := Addr (a);*     *{Address of the memory location}*

Valid operands are only variables, procedures and functions because only these have a physical address. The result is a **typed** pointer. After the operation 'p' contains the address of 'a'. The target is usually a pointer.



## 4.11 System Library - Bit Processing

Bit processing is an essential part of controller applications. Pascal only knows the bit-mask, e.g. *if (x and 1) > 0 then ..*

The remedy is the well-known bit processing of Modula-2 *incl(bit)*, *excl(bit)* and *bit(bit)*. For simplification the *TYPE BIT* was introduced (see also the description of *type bit* above). This type can be used for all 5 Bit-functions (Incl, Excl, Toggle, SetBit, Bit).

The declaration of a bit-variable **always** consists of two levels: first the memory location of the variable, in which the bit is located, has to be defined. This happens with a normal VAR-statement:

```
var Leds[$05] : byte;      or
    bits16   : word;
```

The **type of the memory location** (byte, word) determines if 8 or 16 bits are at the disposal of the programmer. After the declaration of the common variable follows the actual bit-declaration. Thereby the first parameter indicates a memory location, the second parameter indicates the corresponding bit at this location.

```
Const LedBit2 = 3;
var   port6[6] : byte;
      Led2[@port6, Led2Bit] : bit;
```

```
if BIT(Led2) then ...
Toggle(Led2);
```

Bits also can be dynamically generated within the program.

```
Toggle (Leds, 3);      {Bit3 innerhalb 8 Bits}
Incl (bits16, 12);     {Bit12 innerhalb 16 Bits}
```

Further bits can be accessed as usual with 8051 tools in the way "variable.bit".

```
PortB.0:= true;
bool:= PortA.5;
Toggle(PortC.2);
```

"BIT" must be a constant in the range of 0..7.  
This works also with local variables in procedures/functions.

With all bit-write-operations it is also possible to use a number "0" or "1" instead of "False" or "True".

```
PortB.0:= 0;
BitX:= 1;
```

### 4.11.1 INCL

Set Bit

```
Incl (port6, 3);      {This instruction is identical with}
Incl (Led2);          {this instructions provided above definition}
```

Under certain conditions using the procedure **SetBit** is preferred.

## INCL with BitSets:

**Procedure** *Incl* (*SrcDest* : *BitSet*; *op* : *BitSet*);

Where "SrcDest" is the BitSet to change and "op" contains the bits which must be changed.

"op" can be either a BitSet variable of the same type or a BitSet constant "[aa, bb, cc, ...]" of the same type.

## 4.11.2 EXCL

Reset Bit

*Excl* (*Leds*, *a*);                    {Leds is a byte-variable}  
*Excl* (*b*);                            {Symbol b is a Bit declaration}

Under certain circumstances using the procedure **SetBit** is preferred.

## EXCL with BitSets:

**Procedure** *Excl* (*SrcDest* : *BitSet*; *op* : *BitSet*);

Where "SrcDest" is the BitSet to change and "op" contains the bits which must be changed.

"op" can be either a BitSet variable of the same type or a BitSet constant "[aa, bb, cc, ...]" of the same type.

## 4.11.3 TOGGLE

Switch Bit

*Toggle* (*Leds*, 3);  
*Toggle* (*Led2*);

## TOGGLE with BitSets:

**Procedure** *Toggle* (*SrcDest* : *BitSet*; *op* : *BitSet*);

Where "SrcDest" is the BitSet to change and "op" contains the bits which must be changed.

"op" can be either a BitSet variable of the same type or a BitSet constant "[aa, bb, cc, ...]" of the same type.

## 4.11.4 SETBIT

Set/Reset Bit

*SetBit* (*BitType*, *boolean*);

SetBit sets, depending on the parameter boolean, the bit, which is described in the first parameter, to 1 or 0.

There is an access to 8 or 16 bits, depending on the used variable.

To find the definition of the type BIT see also *type bit*.



# AVRco Compiler-Manual

SetBit is a combination of *Incl(bit)* and *Excl(bit)*. So it is **slower** and essential **bigger** (with the number of assembler commands) as a singular *Incl(bit)*, for example. SetBit should only be used, if speed and/or program size are necessary, to replace the following construction:

```
if boolean then  
  Incl (bit);  
else  
  Excl (bit);  
endif;
```

Here it is better faster and shorter:

```
SetBit (bit, boolean);
```

but instead of the following construction:

```
SetBit (bit, true);
```

it is better to use this construction:

```
Incl (bit);
```

```
const  a = 7;  
        c = 5;
```

```
var    v1 = byte;  
        Led1 = [@V1, a];
```

```
Function TestB : boolean;  
begin  
  ...  
  return(x);  
end;
```

```
SetBit (v1, c, TestB);  
SetBit (Led1, TestB);
```

## **SETBIT with BitSets:**

```
Procedure SetBit (SrcDest : BitSet; op : BitSet|Enum; bool : boolean);
```

Where "SrcDest" is the BitSet to change and "op" contains the bits which must be changed.

"op" can be either a BitSet variable of the same type or a BitSet constant "[aa, bb, cc, ...]" of the same type.

Also the basic underlying enumeration can be used.

```
SetBit (myBitSet, [a, b, c], true);  
SetBit (myBitSet, myBitSet1, true);  
SetBit (myBitSet, a, false);
```

### 4.11.5 BIT

Test Bit

```
if Bit (a, 0) then ... ; endif;  
if Bit (b) then ... ; endif;
```

## 4.12 System Library - Diverse System Functions

### 4.12.1 SYSTEM\_RESET

Resets and restarts the whole system. The effect is almost the same as a Hardware Reset, but here some bits within control registers assume a certain value, this is not the case within a system reset. Here interrupts are disabled, followed by a jump to the beginning of the program

If the special function **System\_Init** has been defined, it is naturally executed, too.

*System\_Reset;*

### 4.12.2 DELAY

#### 4.12.2.1 mDelay

Software Delay in msec. The procedure returns after a delay of x msec. The passed parameter has to be between 1 and 65000 (word). Depending on the system the accuracy is about +/-20%. To get an accurate timing **SysTimer** or **SysTimer8** should be used.

**Sleep** should be used in MultiTasking systems, so there is no wastage of runtime.

*Procedure mDelay (d : word);*

*mDelay (100);                    {wait for 100 msec}*

#### 4.12.2.2 uDelay

Software Delay in microseconds x 10. The procedure returns after a passing of n x 10 microseconds. The passed parameter has to be between 1 and 255 (byte). Depending on the system the accuracy is about +/-20%. With CPU-Clocks lower than 2MHz the accuracy is extremely poor.

To get an accurate timing the type **SysTimer** or **SysTimer8** should be used.

*Procedure uDelay(d : byte);*

*uDelay(10);                    {waiting for 100 usec}*

#### 4.12.2.3 uDelay\_1

Software Delay in 1 usec. "uDelay\_1" works precisely only if the CPU Clock is >= 8MHz. If interrupts occur while the Delay is running, the resulting time is stretched, possibly dramatically

*Procedure uDelay\_1 (d : byte);*

#### 4.12.2.4 sDelay

Software Delay in CPU cycles. The procedure returns after a passing of n CPU cycles. The passed parameter has to be between 1 and 255 (byte). An accuracy can not be defined. Useable for very short delays in the micro second range.

*Procedure sDelay (d : byte);*

*sDelay (10);                    {wait for about 10 cycles}*



## 4.12.3 SYSTIMER

### 4.12.3.1 SetSysTimer

This procedure loads a **SysTimer** with the passed value. Therefore the interrupt must be disabled and then enabled again. Because of this a continuous writing to a Systimer should be avoided.

```
var Timer1 : SysTimer8;           {variable of type SysTimer 8bit}
```

```
SetSysTimer (Timer1, 50);
```

Also for SysTimer in UpCount mode.

### 4.12.3.2 SetSysTimerM

*Procedure SetSysTimerM (tm : SysTimer; time : byte|word);*

Define the timeout in milli second values. The internal resolution is still in SysTick values.

This means that with a SysTick of 10msec a parameter value of 27 results in a timer value of 30msec. (TimerValue:= time div SysTickTime).

With a SysTimer8 also floating point SysTick values are allowed, for example 2.5.

The result of the function (in SysTicks) must not exceed 255 ticks.

With the SysTimer (16bit) floating point SysTick times are rounded

### 4.12.3.3 GetSysTimer

This function reads out a **SysTimer** and returns it's actual value. Therefore the interrupt must be disabled and then enabled again. Because of this a continuous writing to a Systimer should be avoided.

```
var Timer1 : SysTimer;           {variable of type SysTimer 16bit}
```

```
Ww:= GetSysTimer (Timer1);
```

Also for SysTimer in UpCount mode.

### 4.12.3.4 ResetSysTimer

This procedure resets a **SysTimer** to a zero value. Therefore the interrupt must be disabled and then enabled again. Because of this a continuous writing to a Systimer should be avoided.

```
var Timer1 : SysTimer8;           {variable of type SysTimer 8bit}
```

```
ResetSysTimer (Timer1);
```

Also for SysTimer in UpCount mode.

## 4.12.3.5 IsSysTimerZero

If SystemTick is imported and a **SysTimer** is defined, then a timer can be polled for zero. Because with each normal access to a timer the interrupt must be disabled and enabled again, which leads to a waste of CPU power and slows down the interrupt system, a better way is to poll the state of such a timer with the function "IsSysTimerZero". This is much faster and interrupts aren't disabled. If the timer = 0 the function returns a true, otherwise a false.

```
var Timer1 : SysTimer;           {variable of type SysTimer 16bit}
```

```
SetSysTimer (Timer1, 50000);
```

```
repeat until isSysTimerZero (Timer1);
```

Not for SysTimer in UpCount mode.

## 4.12.4 LOWER

The function returns the lower of two values. The types of both arguments have to be identical, for example Byte, Int8, word, integer, float.

```
Function Lower (x, y : type) : type;
```

```
x := lower (y, z);
```

## 4.12.5 HIGHER

The function returns the higher of two values. The types of both arguments have to be identical, for example Byte, word, Int8, integer, float.

```
Function Higher (x, y : type) : type;
```

```
x := higher (y, z);
```

## 4.12.6 WITHIN

Function, checks against bounds:

The function checks a number against two bounds. The first value is the lower limit, the second value is the value, which has to be checked, and the third is the higher limit. If the second value is within these limits, so it is the result. If it is too low, the result is the lower limit, otherwise it is the higher limit. All types have to be identical, for example Byte, Int8, word, integer, float.

```
Function Within (lo, x, hi : type) : type;
```

```
x := Within (low, a, high);
```

## 4.12.7 VAL

In standard Pascal the procedure *Val* converts a string-value into its numeric statement.

This general procedure is very complex and would need much code space in flash.

For that reason *val* is in AVRco Pascal **not** implemented but was replaced by several shorter procedures. See chapter "System Library –Standard, Character and String Functions".



## 4.12.8 Block Functions

### 4.12.8.1 FILLBLOCK

Fills a memory area with a byte or char. **P** can be any pointer. A check of the area for validity does not take place. **Fill** can be a byte or a char.

*Procedure FillBlock p : pointer; cnt : word; fill : byte);*

*FillBlock (@Start, len, fill);*

*FillBlock (@array1, SizeOf (array1), 0);*

*FillBlock (@w1, 2, '1');*

### 4.12.8.2 FILLRANDOM (\*4\*)

Fills a memory area with a random value. **P** can be any pointer. A check of the area for validity does not take place. The Random function must be imported from the system.

*Procedure FillRandom (p : pointer; cnt : word);*

*FillRandom (@Start, len!);*

*FillRandom (@array1, SizeOf (array1));*

*FillRandom (@w1, 2);*

### 4.12.8.3 COPYBLOCK

Copies a memory area to another. The blocks should not overlap. A check of the area for validity does not take place. Source and dest can be any pointer.

*Procedure CopyBlock (Source, Dest, len : word);*

*CopyBlock (@array1, @array2, SizeOf (array1));*

*CopyBlock (@w1, @w2, 2);*

### 4.12.8.4 COMPAREBLOCK

Two memory areas can be compared for equal/same content. If the content is the same a true is returned, otherwise a false.

*Function CompareBlock (const addr1, addr2 : pointer; const len : word) : boolean;*

Mögliche Bereiche:

RAM <--> RAM

EEProm <--> EEPROM

ROM <--> RAM

ROM <--> EEPROM

RAM <--> EEPROM



## 4.12.9 Pointer Access Outside the Linear Adress Range

Pointers are always 16 bit und there is no information about the memory area where they should point to. So a pointer implies always an access into the linear CPU address range \$0000..\$FFFF. To access the Flash, the EEPROM or a Banked Device with a pointer the following functions must be used:

### 4.12.9.1 FlashPtr

Redirects the access to the Flash

**Function** *FlashPtr (p:pointer): pointer;*

*Ptr1 := @FlashByte;*  
*bb := FlashPtr (Ptr1)^;*

### 4.12.9.2 EEPROMPtr

Redirects the access to the EEPROM

**Function** *EEPROMPtr (p:pointer): pointer;*

*Ptr1 := @EEPROMByte;*  
*bb := EEPROMPtr (Ptr1)^;*

### 4.12.9.3 UsrDevPtr

Redirects the access to the User Device

**Function** *UsrDevPtr (p:pointer): pointer;*

*Ptr1 := @UsrDevByte;*  
*bb := UsrDevPtr (Ptr1)^;*

### 4.12.9.4 BankDevPtr

Redirects the access to the Banked Device

**Function** *BankDevPtr (b:byte; p:pointer): pointer;*

*Ptr1 := @BankDevByte;*  
*bb := BankDevPtr (2, Ptr1)^;        //bank #2*

## 4.12.10 FLUSHBUFFER

Deletes the content of the buffer of a serial interface. As argument *RxBuffer*, *TxBuffer*, *RxBuffer1*, *TxBuffer1*, *RxBuffer2*, *TxBuffer2*, *RxBuffer3*, *TxBuffer3* are possible.

**Procedure** *FlushBuffer (Buffer : tBuffer);*

*FlushBuffer (RxBuffer);*

## 4.12.11 CRC Checksum

### 4.12.11.1 CRC CHECK

A block of data with the appended CRC-word results in a zero, if this block is again checked by this CRC-function



# AVRco Compiler-Manual

**Function** *CRCcheck* (*p* : pointer; *count* : word) : word;

## **Notes:**

This conform to the CCITT Standard:  $CrcCCITT = x^{16} + x^{12} + x^5 + 1$  with a seed value of \$0810. The programmer can build CRC from RAM, EEPROM and FLASH. It is required that the passed pointer to the memory is defined with the address-of-operator @:

*xx:= CRCcheck (@EEProm, sizeof (EEProm));*

### **4.12.11.2 CRC STREAM**

A CRC stream can be used to repeatedly add bytes/chars of an Input/Output stream into a CRC sum and finally get a 16bit CRC sum calculated over the whole stream.

## **Implementation**

The driver must be imported:

*Import SysTick, CRCstream, ...*

## **Procedures and Functions**

The driver provides three functions:

**Procedure** *CRCstreamInit* (*seed* : word);

This function clears the CRC start value to 0 and "seed" is the initial operating value, in most cases it should be \$0810.

After the call of this init function the byte/char stream can be continuously added to the checksum with

**Function** *CRCstreamAdd* (*value* : byte) : word;

This function adds the byte always returns the current CRC checksum.

**Function** *CRCstreamAddP* (*ptr* : pointer; *count* : word) : word;

This function adds a memory block (RAM) always returns the current CRC checksum.

### **4.12.11.3 FLASH CHECKSUM**

The Compiler offers two 16bit checksum calculations at compile-time which can be checked at runtime. The destination address of the checksum must be defined:

**Define** *FlashChkSum* = \$1ffe; // Byte address

or

**Define** *FlashChkSum* = ProgEnd; // Program end

The checksum will be placed into the absolute address location or at the end of the program. The area of the check starts at address \$0000 and continues up to either the program end or up to the destination address. The DEFINE must be used to set the upper checklimit. Please note that the boot area if present can not be checked for several reasons so the check must end below this area. The number counts in bytes.

The function

**Function** *CalcFlashCheck* : boolean;

generates at runtime checksum and compares it with the value stored in the Flash. The result of this compare is returned as ok/true or failed/false.

Note:

This function is not applicable with devices > 128kB Flash. Furthermore the interrupts are disabled while this function runs.

(\*4\*) In order to support also devices with more than 128kB and also to limit the time the interrupts are disabled the following function should be used:

**Function** *CalcFlashCheck\_S(count : word) : byte;*

This is a so called sequential or partial flash check. It must be called repeatedly until the result is non zero. So the check can be splitted into any partitions and the interrupts are only disabled for a short time. The driver must be imported with:

**From System Import** *FlashCheck\_S;*

The function parameter defines the amount of bytes to be checked.

The function returns a zero if the end is not reached yet. If finished a non-zero result is returned:

1 = check finished and ok  
2 = check finished but failed.

After a non zero value is returned the check parameters are setup to default values so that a check can be restarted.

Example:

```
repeat  
  bb:= CalcFlashCheck_S($1000);  
until bb <> 0;
```

If a **BootBlock** is defined a separate boot checksum can be imported:

**From System Import** *FlashCheck\_B;* (\*4\*)

Then it is possible to execute an extra Flash check over the entire boot area:

**Function** *CalcFlashCheck\_B: boolean;*

#### 4.12.11.4 EEPROM CHECKSUM

Calculates the sum of all bytes in the passed area and returns the negated result.

**Function** *calcChecksum (const start, end : pointer) : word;*

```
{$EEPROM}  
structconst  
  eInt   : word = 1;  
  eStr   : string = 'eeprom';  
  eWord  : word = $1234;  
  eByte  : byte  = $AA;  
  ...  
check:= CalcChecksum (@eStr, @eByte);
```

The address of "eByte" is used as the end-pointer but the value of "eByte" itself is not included into the calculation.

It's also possible to build an EEprom checksum at compile-time and store it into the EEprom as a structured const:



# AVRco Compiler-Manual

*{\$EEPROM}*

**structconst**

```
eInt : word = 1;  
eStr : string = 'eeprom';  
eWord : word = $1234;  
eByte : byte = $AA;  
eCheck : word = CalcChecksum (@eStr, @eByte);
```

The memory location "eCheck" now contains the checksum of all EEprom bytes from "eStr" (inclusive) to "eByte" (exclusive). A special construction allows to use the destination of the function result as the end pointer with using the "\$" sign:

```
eCheck : word = CalcChecksum (@eStr, $);
```

Both operations are only valid if placed in the "StructConst" area of the EEprom.

## 4.12.12 RANDOM

Function, returns a random number of the type Word. Random must be imported.

*From System import Random...*

**Function** *Random* : word;

*W:= Random;*

## 4.12.13 RANDOMRANGE (\*4\*)

Function, returns a random number of the type Word. The value is limited by min and max. Random must be imported.

*From System import Random...*

**Function** *RandomRange*(min, max : word) : word;

*W:= RandomRange(100, 500);*

## 4.12.14 SQR, (\*4\*)(\*P\*): SQR\_D

Functions, return the square of the float/double argument

**Function** *Sqr* (f : float) : float;

**Function** *Sqr\_D* (f : double) : double;

*f:= Sqr (f);*

## 4.12.15 SQRT, (\*4\*)(\*P\*): SQRT\_D

Functions, return the square root of a float/double argument

**Function** *Sqrt* (f : float) : float;

**Function** *Sqrt\_D* (f : double) : double;

*f:= Sqrt (f);*

## 4.12.16 POW, (\*4\*)(\*P\*): POW\_D

Functions, return the result of x power of y. The base (x) must always be positive!

**Function** *Pow* (x, y : float) : float;

**Function** *Pow\_D* (const x, y : double) : double;

*f:= Pow (x, y);*

## 4.12.17 POW10, (\*4\*)(\*P\*): POW10\_D

Functions, return the result of 10 power of x

**Function** *Pow10* (*f* : float) : float;

**Function** *Pow10\_D*(const *x* : double) : double;

*f* := *Pow10* (*x*);

## 4.12.18 EXP, (\*4\*)(\*P\*): EXP\_D

The functions *Exp* and *Exp-D* return the power of X (float).

The return value is e power of X, and e is the base of the natural logarithm.

**Function** *Exp* (*f* : float) : float;

**Function** *Exp\_D* (const *f* : double) : double;

*f* := *Exp* (*x*);

## 4.12.19 LogN, (\*4\*)(\*P\*): LogN\_D

The functions *LogN* and *LogN\_D* return as result the nature logarithm of X.

The log naturalis is based on the Euler number  $e = 2.71\dots$

**Function** *LogN*(*f* : float) : float;

**Function** *LogN\_D*(const *f* : double) : double;

*f* := *LogN*(*x*);

## 4.12.20 Log10, (\*4\*)(\*P\*): Log10\_D

The functions *Log10* and *Log10\_D* return as result the logarithm to base 10 of X.

**Function** *Log10* (*f* : float) : float;

**Function** *Log10\_D* (const *f* : double) : double;

*f* := *Log10* (*x*);

## 4.12.21 Trigonometrical Functions

### 4.12.21.1 TAN, (\*4\*)(\*P\*): TAN\_D

The functions return as result the tangent of an angle  $w$  (in radians).

**Function** *Tan* ( $w : float$ ) : float;  
**Function** *Tan\_D* ( $const f : double$ ) : double;

$t := Tan(w);$

### 4.12.21.2 TAND, (\*4\*)(\*P\*): TAND\_D

The functions return as result the tangent of the angle  $w$  (in degrees).

**Function** *TanD* ( $w : float$ ) : float;  
**Function** *TanD\_D* ( $const f : double$ ) : double;

$t := TanD(w);$

### 4.12.21.3 ARCTAN, (\*4\*)(\*P\*): ARCTAN\_D

The functions return the arctangent in radians.

**Function** *ArcTan* ( $w : float$ ) : float;  
**Function** *ArcTan\_D* ( $const f : double$ ) : double;

$a := ArcTan(w);$

### 4.12.21.4 SIN, (\*4\*)(\*P\*): SIN\_D

The functions return the sine of the argument.  
 $w$  is a term of the type float/double. Return the sine of the angle  $w$  in radians.

**Function** *Sin* ( $w : float$ ) : float;  
**Function** *Sin\_D* ( $const w : double$ ) : double;

$s := Sin(w);$

### 4.12.21.5 SININT

The function returns the Sine of the angle multiplied by the Integer argument.  
 Very fast and short!!

**Function** *SinInt* ( $angle, v : integer$ ) : integer;

### 4.12.21.6 SININT16

This function calculates the sine of an angle, multiplies the result by 10000 and returns the result as an integer value. The angle must be passed in tenths of a degree (2.5deg -> 25).  
 The result for the angle 90deg (parameter angle = 900) is then 10000.

**Function** *SinInt16* ( $angle : integer$ ) : integer;      // angle in 0.1deg  
 // result:= round (Sin (angle / 10) \* 10000);

Provides much more precise results without needing much more runtime.  
 Disadvantage of this version:  
 there must be a sine table in ROM/Flash of about 2kByte size.

## 4.12.21.7 SIND, (\*4\*)(\*P\*): SIND\_D

The functions return the sine of the argument.  
w is a term of the type float/double. Return the sine of the angle w in degrees.

**Function** *SinD* (w : float) : float;  
**Function** *SinD\_D* (w : double) : double;

s := *SinD* (w);

## 4.12.21.8 COS, (\*4\*)(\*P\*): COS\_D

The functions return the cosine of the argument.  
w is a term of the type float/double. Return the cosine of the angle w in radians.

**Function** *Cos* (w : float) : float;  
**Function** *Cos\_D* (w : double) : double;

c := *Cos* (w);

## 4.12.21.9 COSINT

The function returns the Cosine of the angle multiplied by the Integer argument.  
Very fast and short!!

**Function** *CosInt* (angle, v : integer) : integer;

## 4.12.21.10 COSINT16

This function calculates the cosine of an angle, multiply the result by 10000 and return the result as an integer value. The angle must be passed in tenths of a degree (2.5deg -> 25).  
The result of the angle 45deg (parameter angle = 450) for the cosine then is 7071.

**Function** *CosInt16* (angle : integer) : integer; // angle in 0.1deg  
// result := round (Cos (angle / 10) \* 10000);

Provides much more precise results without needing much more runtime.  
Disadvantage of this version:  
there must be a sine table in ROM/Flash of about 2kByte size.

## 4.12.21.11 COSD, (\*4\*)(\*P\*): COSD\_D

The functions return the cosine of the argument.  
w is a term of the type float/double. Return the cosine of the angle w in degrees.

**Function** *CosD* (w : float) : float;  
**Function** *CosD\_D* (w : double) : double;

c := *CosD* (w);



## 4.12.21.12 DEG TORAD, (\*4\*)(\*P\*): DEG TORAD\_D

The functions turn an angular value, which is in degrees, into radians.  
360 degrees corresponds to 2 Pi radians.  
{ rad := degree \* Pi / 180 }

**Function** *DegToRad* (*w* : float) : float;  
**Function** *Deg2Rad\_D* (const *f* : double) : double;

*r* := *DegToRad* (*w*);

## 4.12.21.13 RADTODEG, (\*4\*)(\*P\*): RADTODEG\_D

The functions turn an angular value in radians into degrees.  
{ degree := rad \* 180 / Pi }

**Function** *RadToDeg* (*w* : float) : float;  
**Function** *Rad2Deg\_D* (const *f* : double) : double;

*w* := *RadToDeg* (*r*);

## 4.12.21.14 ROTATEPNTI

The point(XPo, YPo) is rotated by angle (degrees). The result is returned in XPd, YPd.

**Procedure** *RotatePnti* (*angle*, *XPo*, *YPo* : integer; **var** *XPd*, *YPd* : integer);

## 4.12.22 TRUNC, (\*4\*)(\*P\*): TRUNC\_D

The functions truncate a value of the type Float/Double to an Integer value.  
*f* is a term of the type Float/Double. Trunc returns the type (integer or longint) corresponding to the destination.

**Function** *Trunc* (*f* : float) : integer; {LongInt}  
**Function** *Trunc\_D* (const *f* : double) : Integer [LongInt];

*i* := *Trunc* (*f*);

## 4.12.23 ROUND, (\*4\*)(\*P\*): ROUND\_D

The functions round a value of the type Float/Double to a value of the type integer (Byte, Word, Longint, Longword).  
*f* is a term of the type float/double. Return an int-value, which describes the value of *f* rounded to the next integer number. If *f* is exactly in the middle of two integer numbers, in the result the number with the highest absolute value is returned.

**Function** *Round* (*f* : float) : integer; {Byte, Word, LongInt, LongWord}  
**Function** *Round\_D* (const *f* : double) : integer [LongInt];

*i* := *Round* (*f*);



## 4.12.24 FRAC, (\*4\*)(\*P\*): FRAC\_D

The functions return the fractional part of the argument of x. X is a term of the type float/double. The result is the fraction part of x; i.e.  $\text{Frac}(x) = x - \text{Int}(x)$ .

**Function** *Frac* (*f* : float) : float;

**Function** *Frac\_D* (const *f* : double) : double;

*f* := *Frac* (*x*);

## 4.12.25 INT, (\*4\*)(\*P\*): INT\_D

The functions return the integer part of the argument. X is a term of the type Float/Double. The result is the integer part of X; i.e., X is rounded to 0.

**Function** *Int* (*f* : float) : float;

**Function** *Int\_D* (const *f* : double) : double;

*f* := *Int* (*x*);

## 4.12.26 GETTABLE

The function GetTable returns a member of a LookUp-Table.

**Function** *GetTable* (*t* : Table; *index* : byte) : type;

*x* := *GetTable* (*Table1*, *b*);

## 4.12.27 SETTABLE

The procedure SetTable changes a member in a LookUp-Table.

**Procedure** *SetTable* (*t* : Table; *index* : byte; *new* : type);

*SetTable* (*Table1*, *b*, *x*);

## 4.12.28 Conversion to Strings

With all this kinds of conversion do not underestimate the frame usage. This is also true for Processes and Tasks where these functions are used.

e.g.:

ByteToStr > 12 bytes

IntToStr > 17 bytes

LongToStr > 37 bytes !

Int64ToStr > 70 bytes !!!

## 4.12.28.1 BYTETOSTR

Converts a numeric 8bit value into a string. The parameter can be an ordinal numeric constant (byte, Int8, Enum) or a variable of this type.

**Function** *ByteToStr* (*b* : byte[Int8, Enum]) : string;

```
const st = '1234' + 'R' + #7 + ^L;
var st1 : string[9];
    bb : byte;
```

```
write (LCDout, ByteToStr (100));
write (SERout, ByteToStr (100:6)); {-> ' 100'}
bb:= 10;
st1:= ByteToStr (bb);           {-> '10'}
st1:= ByteToStr (bb:6);        {-> ' 10'}
st1:= ByteToStr (bb:6:1);      {-> ' 1.0'}
st1:= ByteToStr (bb:6:1:'_');  {-> ' ____1.0'}
st1:= ByteToStr (bb:6:'_');    {-> ' ____10'}
```

## 4.12.28.2 INTTOSTR

Converts a numeric 16bit value into a string. The parameter can be an ordinal numeric constant (integer, word) or a variable of this type.

**Function** *IntToStr* (*i* : word) : string;

```
write (LCDout, IntToStr (100));
write (SERout, IntToStr (i:6:2)); {-> ' 1.00'}
st1:= IntToStr (123:4);          {-> ' 123'}
ii:= -1;
st1:= IntToStr (ii);             {-> '-1'}
st1:= IntToStr (ii:10);          {-> ' -1'}
st1:= IntToStr (ii:10:2);        {-> ' -0.01'}
st1:= IntToStr (ii:10:2:'x');    {-> 'xxxxx-0.01'}
st1:= IntToStr (ii:10:'x');      {-> 'xxxxxxxx-1'}
```

## 4.12.28.3 LONGTOSTR

Converts a numeric 32bit value into a string. The parameter can be an ordinal numeric constant (Longint, Longword) or a variable of this type.

**Function** *LongToStr* (*ii* : longword) : string;

```
write (LCDout, LongToStr (100000));
write (SERout, LongToStr (ii:6:2)); {-> ' 1.00'}
st1:= LongToStr (123456:8);       {-> ' 123456'}
Li:= 100;
st1:= IntToStr (Li);              {-> '100'}
st1:= IntToStr (Li:10);           {-> ' 100'}
st1:= IntToStr (Li:10:2);         {-> ' 1.00'}
st1:= IntToStr (Li:10:2:'x');     {-> 'xxxxxx1.00'}
st1:= IntToStr (Li:10:'x');       {-> 'xxxxxxx100'}
```



# AVRco Compiler-Manual

## 4.12.28.4 FLOATTOSTR

Converts a Floating Point value into a string. The parameter can be a float constant or a variable of this type.

**Function** *FloatToStr* (*f* : float) : string;

```

write (LCDout, FloatToStr (1000.00));
write (SERout, FloatToStr (f:6:2));    {-> ' 1.00'}
st1:= FloatToStr (123.456:8);         {-> ' 123'}
f:= -100.1;
st:= FloatToStr (f);                  {-> '-100.1'}
st1:= FloatToStr (f:11);               {-> ' -100'}
st:= FloatToStr (f:11:0);              {-> ' -100'}
st:= FloatToStr (f:11:2);              {-> ' -100.10'}
st:= FloatToStr (f:11:2:'=');          {-> '====-100.10'}
st1:= FloatToStr (f:11:'=');           {-> '=====-100'}
st1:= FloatToStr (f:'E');              {-> '-1.001E2'}
st1:= FloatToStr (f:'E':11);           {-> ' -1.001E2'}
st1:= FloatToStr (f:'E':11:'=');       {-> '====-1.001E2'}

```

If no formatting is given "FloatToStr(f)" then the resulting string contains all digits.

If there is one parameter given "FloatToStr(f:n)" then the length of the string is at least "n" characters, if necessary with leading spaces. Digits after the decimal point are discarded. It is the same as with "FloatToStr(f:n:0)"

If two parameters are given the second one can either be numeric or a character. If numeric it defines the count of the digits after the decimal point. If it is a character it defines the fill character and there are no digits and no decimal point.

- a. FloatToStr(f:n:3) returns a string with the total length of "n" with 3 digits after the decimal point. Leading digits are filled with spaces.
- b. FloatToStr(f:n:'x') returns a string with the total length of "n" where leading spaces are replaced by an "x". No decimal digits.

If 3 parameters are given "FloatToStr(f:n:k:'x')" a string with the total length of "n" with "k" decimals is returned. Leading spaces are replaced by 'x'

### **Attention:**

This system function requires additional 20 bytes on the frame.

### **Note:**

The functions *ByteToStr*, *IntToStr*, *LongToStr* and *FloatToStr* accept also byte variables for the digits and decimal parameter.

But please note that range checking by the compiler of these parameters not possible. The programmer himself is responsible for valid values. With illegal parameter unexpected results can occur. Even a system crash is possible

### Example for variable parameters:

```

var digs,
    dec : byte
    ii  : integer;
...
st:= IntToStr (ii:digs:dec);

```

## 4.12.28.5 BOOLTOSTR

Converts a boolean value to it's corresponding string value.

Two functions are implemented. The first converts the boolean argument into the string 'true' or 'false'. The second function returns either the TrueStr or FalseStr, dependant of the value of the boolean argument.

**Function** BoolToStr (*bool* : boolean) : string;  
**Function** BoolToStr (*bool* : boolean; *TrueStr*, *FalseStr* : string) : string;

#### 4.12.28.6 LONG64TOSTR (\*4\*)(\*P\*)

Converts a numeric 64bit value into a string. The parameter can be an ordinal numeric constant (Int64, Word64).

**Function** Long64ToStr (**const** *ii* : Int64|Word64; **const** *len* : byte[: **const** *space* : char]] : string;

#### 4.12.28.7 BYTETOHEX

Converts a numeric 8bit value into a hex-string. The parameter can be an ordinal numeric constant (byte, Int8, Enum) or a variable of this type.

**Function** ByteToHex (*b* : byte[Int8, Enum]) : string;

```
x:= 48;  
st1:= ByteToHex (x) + 'h';           {st1 contains '30h'}
```

#### 4.12.28.8 INTTOHEX

Converts a numeric 16bit value into a hex-string. The parameter can be an ordinal numeric constant (integer, word) or a variable of this type.

**Function** IntToHex (*i* : integer) : string;

```
st1:= IntToHex (123);                {st1 contains '7B'}
```

#### 4.12.28.9 LONGTOHEX

Converts a numeric 32bit value into a hex-string. The parameter can be an ordinal numeric constant (Longint, Longword) or a variable of this type.

**Function** LongToHex (*w* : longword) : string;

```
st1:= LongToHex (123456);            {st1 contains '1E240'}
```

#### 4.12.28.10 LONG64TOHEX (\*4\*)(\*P\*)

Converts a numeric 64bit value into a hex-string. The parameter can be an ordinal numeric constant (Int64, Word64).

**Function** Long64ToHex (**const** *ii* : Int64|Word64) : string;

#### 4.12.28.11 BYTETOBIN

**Function** ByteToBin (*value* : byte[Int8]) : string;

The result of the functions is the representation of the bits of the arguments by '0' or '1' in the resulting string. A byte with the value 5 then results in '00000101'.

#### 4.12.28.12 INTTOBIN



# AVRco Compiler-Manual

**Function** *IntToBin* (*value : word|integer*) : *string*;

The result of the functions is the representation of the bits of the arguments by '0' or '1' in the resulting string. A word with the value 257 then results in '0000000100000001'.

## 4.12.29 BYTETOBCD

**Function** *ByteToBCD* (*b : byte*) : *byte*;

Real Time Clocks frequently use what is known as *Packed BCD* values:

BCD numbers represented in the highest 4bits are the tens digits and in the lowest 4 bits ones digits.

Each digit can only represent a number of 0..9.

This function converts a byte value into the BCD format. Please note that max. number that can be passed to this function is 99. A value of \$10 (16dec) is returned as \$16.

## 4.12.30 WORDTOBCD

**Function** *WordToBCD* (*w : word*) : *word*;

This function converts a word value into the BCD format. Please note that max. number that can be passed to this function is 9999. A value of \$270F (9999dec) is returned as \$9999.

In the highest 4bits are the thousands digits, in the next 4bits the hundreds digits, then the tens digits and in the lowest 4 bits the ones digits. Each digit can only represent a number of 0..9.

## 4.12.31 BCDTOBYTE

**Function** *BCDtoByte* (*b : byte*) : *byte*;

Real Time Clocks frequently use what is known as *Packed BCD* values:

BCD numbers represented in the highest 4bits are the tens digits and in the lowest 4 bits ones digits.

Each digit can only represent a number of 0..9.

This format will be read out.

The function expects a packed BCD value and converts it to a decimal Byte \$16 -> \$10 (=16dec).

## 4.12.32 PCU SI-Conversion (\*P\*)

by Tassilo Heinrich

### 4.12.32.1 Utility Functions

**Function** *CountsToVolts* (*Counts:Word; VRef:Float; Res:word; Gain:Float*):*Float*;

Converts AD-Counts to Volts.

*Volt := CountsToVolts (512, 5.0, 1024, 1.0): Float; // = 2.5Volt*

**Function** *MX\_B* (*m:Float; x:Float; b:Float*): *Float*;

$Y = mX + t$

**Function** *ByteToBcd* (*byteVal : Byte*) : *Byte*;

Converts Byte to BCD.

**Function** *BcdToByte* (*bcdVal : Byte*) : *Byte*;

Converts BCD to Byte.

### 4.12.32.2 Temperature

**Function** *F\_CelsiusToKelvin* (*Cel:Float*): *Float*;

Converts °C to Kelvin.

**Function** *F\_KelvinToCelsius* (*Kelvin:Float*): *Float*;

Converts Kelvin to °C.

**Function** *F\_FahrenheitToCelsius* (*Fahrenheit:Float*): *Float*;

Converts Fahrenheit to °C.

**Function** *F\_CelsiusToFahrenheit* (*Celsius:Float*): *Float*;

Converts °C to Fahrenheit.

### 4.12.32.3 Volume

**Function** *F\_LiterToGal* (*Liter:Float*): *Float*;

Converts Litres to US Galons.

**Function** *F\_GalToLiter* (*Gal:Float*): *Float*;

Converts US Galons to Litres.

**Function** *F\_LiterToCuFt* (*Liter:Float*): *Float*;

Converts Litres to Feet<sup>3</sup>.

**Function** *F\_CuFtToLiter* (*CuFt:Float*): *Float*;

Converts Feet<sup>3</sup> to Litres.

**Function** *F\_CuFtToCuln* (*CuFt:Float*): *Float*;

Converts Feet<sup>3</sup> to Inches<sup>3</sup>.

**Function** *CunctiosCulnToCuFt* (*Culn:Float*): *Float*;

Converts Inches<sup>3</sup> to Feet<sup>3</sup>.



# AVRco Compiler-Manual

## 4.12.32.4 Pressure

**Function** *F\_PSItoBar (PSI:Float): Float;*  
Converts PSI to Bar.

**Function** *F\_mBarToPSI (mBar:Float): Float;*  
Converts Bar to PSI.

**Function** *F\_mmHgToBar (mmHg:Float): Float;*  
Converts mmHg to Bar.

**Function** *F\_mBarTommHg (mBar:Float): Float;*  
Converts Bar to mmHg.

**Function** *F\_cmH2OtoBar (cmH2O:Float): Float;*  
Converts cmH<sub>2</sub>O to Bar.

**Function** *F\_mBarTocmH2O (mBar:Float): Float;*  
Converts Bar to cmH<sub>2</sub>O.

## 4.12.32.5 Length

**Function** *F\_MeterToFeet (Meter:Float): Float;*  
Converts Meter to Feet.

**Function** *F\_FeetToMeter (Feet:Float): Float;*  
Converts Feet to Meter

**Function** *F\_InToMeter (Inch:Float): Float;*  
Converts Inches to Meter.

**Function** *F\_cMeterToIn (cMeter:Float): Float;*  
Converts Meter to Inches.

**Function** *F\_ydToMeter (yd:Float): Float;*  
Converts Yards to Meter.

**Function** *F\_MeterToyd (Meter:Float): Float;*  
Converts Meter to Yards.

**Function** *F\_miTokMeter (mi:float): Float;*  
Converts Miles to Kilometer.

**Function** *F\_kMeterToMi (kMeter:Float): Float;*  
Converts Kilometer to Miles.

**Function** *F\_nmiTokMeter (nmi:float): Float;*  
Converts Nautical Miles to Kilometer.

**Function** *F\_kMeterTonmi (kMeter:Float): Float;*  
Converts Kilometer to Nautical Miles.

## 4.12.32.6 Area

**Function** *F\_SqrMeterToSqrFeet (Meter:Float): Float;*



Converts Meter<sup>2</sup> to Feet<sup>2</sup>.

**Function** *F\_SqrFeetToSqrMeter (Feet:Float): Float;*  
Converts Feet<sup>2</sup> to Meter<sup>2</sup>.

**Function** *F\_SqrInToSqrMeter (Inch:Float): Float;*  
Converts Inches<sup>2</sup> to Centimeter<sup>2</sup>

**Function** *F\_SqrMeterToSqrIn (cMeter:Float): Float;*  
Converts Centimeter<sup>2</sup> to Inches<sup>2</sup>

**Function** *F\_SqrydToSqrMeter (yd:Float): Float;*  
Converts Yards<sup>2</sup> to Meter<sup>2</sup>

**Function** *F\_SqrMeterToSqryd (Meter:Float): Float;*  
Converts Meter<sup>2</sup> to Yards<sup>2</sup>

**Function** *F\_SqrmiToSqrkMeter (mi:float): Float;*  
Converts Miles<sup>2</sup> to Kilometer<sup>2</sup>

**Function** *F\_SqrkMeterToSqrmi (kMeter:Float): Float;*  
Converts Kilometer<sup>2</sup> to Miles<sup>2</sup>

## 4.12.32.7 Weight

**Function** *F\_KaratToGramm (Karat:Float): Float;*  
Converts Carat to Gram

**Function** *F\_GrammToKarat (Gramm:Float): Float;*  
Converts Gram to Carat

**Function** *F\_GrainsToGramm (Grains:Float): Float;*  
Converts Grains to Gram

**Function** *F\_GrammToOunces (Gramm:Float): Float;*  
Converts Gram to Ounces.

**Function** *F\_OuncesToGramm (Ounces:Float): Float;*  
Converts Ounces to Gram

**Function** *F\_GrammToOuncesTroy (Gramm:Float): Float;*  
Converts Gram to Ounces troy.

**Function** *F\_OuncesTroyToGramm (OuncesTroy:Float): Float;*  
Converts Ounces troy to Gram.

**Function** *F\_kGrammToStones (kGramm:Float): Float;*  
Converts Kilograms to Stones.

**Function** *F\_StonesTokGramm (Stones:Float): Float;*  
Converts Stones to Kilogram.

**Function** *F\_kGrammToPounds (kGramm:Float): Float;*  
Converts Kilogram to Pound.

**Function** *F\_PoundsTokGramm (Pounds:Float): Float;*  
Converts Pounds to Kilograms.



## 4.12.32.8 Energy

**Function** *F\_kWToPS (kW:Float): Float;*  
Converts Kilowatts to PS.

**Function** *F\_PSToKW (PS:Float): Float;*  
Converts PS to Kilowatts.

**Function** *F\_CalToJ (Cal:Float): Float;*  
Converts Calories to Joule.

**Function** *F\_JtoCal (J:Float): Float;*  
Converts Joule to Calories.

## 4.12.32.9 Integer Functions

**Function** *I\_CelsiusToKelvin (Cel:Integer): Integer;*

**Function** *I\_KelvinToCelsius (Kel:Integer): Integer;*

**Function** *I\_FahrenheitToCelsius (Fahrenheit:Integer): Integer;*

**Function** *I\_CelsiusToFahrenheit (Celsius:Integer): Integer;*

## 4.12.32.10 Constants

ZeroPoint : Float = -273.16;

MSL\_Pressure\_PSI : float = 14.697;

MSL\_Pressure\_InHg : float = 29.92;

MSL\_Pressure\_mBar : float = 1013.25;

Euler : Float = 2.7182818284;

## 4.12.33 Interpolation

Many sensors and other functions show a nonlinear curve. This means that the relation between an input value and the corresponding output value is not linear.

Examples: PT100, PTC, NTC, light-detectors, and also diodes.

This is only a very small count of nonlinear sensors. Normally the measured result has a fixed relation to the external events, but in many cases this relationship is not linear, but logarithmic, cubic etc. A PT100 shows a resistance of 100 Ohms at 0degC, at 50degC 124Ohms and at 100degC 143Ohms. The relation between temperature and the resistance value is nonlinear.

There are two ways to calculate the temperature from the resistance:

1. With a proper formula, which is usually a complex thing, one can calculate the temperature which corresponds with the resistance.
2. Build a so called LookUp-Table. Insert in steps resistance values and the related temperature values. Access the table with the resistance value as an index. The result is the temperature. If the input value ranges from 100 to 200 there must be 100 value pairs in the table. More difficult is a large span of the input values (0 to 1023). Then the table must have 1024 entries

The following implementation is table based, with paired values (search/result).

The LookUp algorithm searches with a known value in the table until either this value is found or this argument fits between two values.

The search is done with a binary search function for best speed results.

If a proper value(s) is found it will be linearly interpolated. This method allows short tables, dependent of the required accuracy. If the count of the value pairs is relative high, the linear interpolation results in an acceptable accuracy.

### 4.12.33.1 InterPolX, InterPolY

**Function** *InterPolX* (**const** LookUp : pointer; x : integer; **var** y : integer) : boolean;

**Function** *InterPolX* (**const** LookUp : pointer; x : longint; **var** y : longint) : boolean;

**Function** *InterPolX* (**const** LookUp : pointer; x : float; **var** y : float) : boolean;

**Function** *InterPolY* (**const** LookUp : pointer; y : integer; **var** x : integer) : boolean;

**Function** *InterPolY* (**const** LookUp : pointer; y : longint; **var** x : longint) : boolean;

**Function** *InterPolY* (**const** LookUp : pointer; y : float; **var** x : float) : boolean;

The pointer must point into a table in the **ROM** or **EEPROM**. The first argument is the search value. The result is placed into the location of the second argument, if the function was successful.

As a support tool for the creation of the lookup table a Table Generator "CurveGen" is included. With it's help one can interactive and graphically create a curve and then store it into a binary file which can be imported into the application.

#### Example Program:

An example program can be found in the directory `..\E-Lab\AVRco\Demos\Interpol`.

Here an optical distance/proximity sensor (Sharp) is sampled by the ADC and then linearised.

A Datasheet can be found in `..\E-Lab\DOCs\Sharp.pdf`

A detailed description of the above functions and also of the support tool "CurveGen" is in the *Tools Manual*.

An schematic example can be found in `..\E-Lab\DOCs\NonLinSensSch.pdf`



## 4.12.34 Moving Average Filter

Build an average for example over the last 16 measure results.

The results are stable values which nevertheless are changing too fast to follow changing input values.

The filter is constructed like an array of Byte/Word/Integer etc. If a new value is inserted, this value replaces the oldest value in the array. By adding all values and then dividing the sum by the value count the resulting average is built.

The filter (Array) must be declared as a variable in standard RAM. Banks, EEprom, procedure-local variables or records etc. are not allowed.

The filter can consist of Bytes, Words, Integer, LongWords, LongInts or Float. The size/count runs from 0 to x - 1, where x is always a power of 2 in the range 4..256. With the declaration the upper limit must be one less than the sample count.

*var Filter : AVfilter[0..15] of integer;*

### 4.12.34.1 PresetAVfilter

*Procedure PresetAVfilter (var Filter : AVfilter; val : type);*

Populates the entire filter with "val".

### 4.12.34.2 SetAVfilter

*Function SetAVfilter (var Filter : AVfilter; val : type) : type;*

Replaces the oldest entry with "val" and returns with the new build average value.

### 4.12.34.3 AddAVfilter

*Procedure AddAVfilter (var Filter : AVfilter; val : type);*

Sometimes it is not necessary that a new average must be build when a new value is inserted.

This function works in the same way as "SetAVfilter" except that no new average value is calculated and returned.

### 4.12.34.4 GetAVfilter

*Function GetAVfilter (var Filter : AVfilter) : type;*

Computes the current averages without changing the content of the filter.

### 4.12.34.5 DeclAVfilter

*Function DeclAVfilter (var Filter : AVfilter) : type;*

Calculates the gradient between the oldest and the youngest entry.

The result is always signed. This means that with a Byte, Integer or Word filter the result is always an integer. With a LongWord or LongInt filter the result is always an LongInt

#### Example Program:

An Example can be found in the directory `..\E-Lab\AVRco\Demos\AVfilter`.

## 4.12.35 Network-Functions

### 4.12.35.1 Predefined Types

**type** TIPAddr = **array**[0..3] of Byte;  
**type** TMACAddr = **array**[0..5] of Byte;

### 4.12.35.2 Converting Functions

**Procedure** STRtoIP (IPstr : String[15]; **var** Result : TIPAddr);  
Converts an IP-address string "aaa:bbb:ccc:ddd" to a byte array

**Procedure** STRtoMAC (MACstr : String[17]; **var** Result : TMACAddr);  
Converts a MAC-address string "aa:bb:cc:dd:ee:ff" to a byte array

**Function** IPtoSTR (IP : TIPAddr) : String[15];  
Converts an IP-address array to a string "aaa:bbb:ccc:ddd"

**Function** MACtoSTR (MAC : TMACAddr) : String[17];  
Converts a MAC-address array to a string "aa:bb:cc:dd:ee:ff"

### 4.12.35.3 Compare Functions

**Function** CompareNet (a1, a2, mask : TIPAddr) : boolean;  
Compares the network part of two IP-address arrays

**Function** CompareIP (ip1, ip2 : TIPAddr) : boolean;  
Compares two IP-address arrays

**Function** CompareMAC (mac1, mac2 : TMACAddr) : boolean;  
Compares two MAC-address arrays

### 4.12.35.4 Miscellaneous Functions

**Procedure** SwapIPAddr (**var** ip : TIPAddr);  
Mirrors an IP-address. Converts A3-A2-A1-A0 to A0-A1-A2-A3.

**Procedure** SwapMACAddr (**var** mac : TMACAddr);  
Mirrors a MAC-address. Converts A5-A4-A3-A2-A1-A0 to A0-A1-A2-A3-A4-A5.

## 4.13 System Library - String Formatting

The destination of a string conversion can be a string variable or the procedure WRITE as shown in the description of *ByteToStr* and *IntToStr* above.

A conversion of values (variables) within strings as well as the formatting the output is necessary to handle, for example, an LCD-display.

The conversion routines in co-operation with WRITE solve this need.

The **Formatting** is almost the same as in Turbo Pascal. The parameter after the **first** colon specifies the required overall length. Be careful, as the string can get longer, but not shorter than the format specified. For example "100:0" results in the string "100", so the length is three not zero.

The parameter after the **second** colon is not identical with TurboPascal within *ByteToStr* and *IntToStr*. This parameter specifies the **number of** the post decimal positions. But integers do not possess any decimal positions! That is true, but often integers with a fixed point are used and then there is the problem to get the correct representation.

Take care, the decimal point is counted, too! Thus "100:6:2" results " 1.00" with two leading spaces, so altogether the length is 6.

Leading spaces can be replaced by another character, which must be defined after the last colon.

### 4.13.1 Decimal Separator

Some of the string conversions above work with a decimal point. This character is by default a "." This constant can be redefined e.g. to a "," to fit for some requirements. For this the following statement must be inserted into the Define Block:

```
Define DecimalSep = ',';
```

### 4.13.2 WRITE

String or number output by a procedure with conversion. The first parameter of Write must be a procedure.

The first parameter is called as a procedure (Device) once for every character in the second parameter, passing each character in turn as a parameter to the first procedure. The procedure **must** have the following form:

```
proc (b : byte);
```

A simple output to external devices is possible, because the procedure can access any hardware.

```
const st = '1234' + 'R' + #7 + ^L;
var st1 : string[5];
```

<i>Write</i> (proc, 'x');	{'x' is returned }
<i>Write</i> (proc, st1);	{st1 is returned }
<i>Write</i> (LCDout, ByteToStr (100));	
<i>Write</i> (LCDout, st[0]);	{Length byte}
<i>Write</i> (SerOut, st);	{Output of compl. string}
<i>Write</i> (LCDout, st[1]);	{1. char in st}
<i>Write</i> (LCDout, '1234');	
<i>Write</i> (SERout, ByteToStr (100:6));	{-> ' 100}
<i>Write</i> (SERout, ByteToStr (100:6:2));	{-> ' 1.00}
<i>Write</i> (SERout, IntToStr (i:6:1));	{-> ' 10.0}
<i>Write</i> (DispOut, #13 + 'Hallo');	

Write is able to access user-defined device-drivers. Pay attention to some special requirements:

This procedure must have a passing parameter of the type byte or Char, it may not have a local variable and hence no frame. Further only the pseudo-accumulators “\_ACCA, \_ACCB, \_ACCCHI and \_ACCCLO = Z” may be used within the procedure. That excludes a calling of other procedures and functions as well as the system functions.

If additional ACCUs for intermediate memory are used anyhow, they have to be saved with help of Push/Pop. These restrictions mean that in practice this procedure must be written completely in assembler. The passing parameter is via \_ACCA. The compiler switch `{$NOFRAME}` is essential

```
{$NOFRAME}  
Procedure TestDriver (const b : byte);  
begin  
  ASM;  
  OUT SPDR, _ACCA;    { SPI Data reg }  
  ENDASM;  
end;
```

### 4.13.3 WRITELN

WriteLn appends a CarriageReturn/LineFeed \$0D+\$0A to the output line.

```
Procedure WriteLn (DeviceFunc : function; var str : string);
```

```
WriteLn (SerOut, 'Test');  
WriteLn (SerOut);           // write an empty line
```

### 4.13.4 READ

Reading of a character or string is similar to write. The first parameter is a function, which returns a character as result. The second parameter is a string variable and the optional third is the number of the bytes, which have to be read, or a delimiter character.

The function Read has three modes of operation:

1. read a string completely. The string gets filled completely
2. read a certain number of characters. The 3<sup>rd</sup> parameter must be of the type **byte**.
3. read bytes/chars until a limiter char appears that must be specified as 3<sup>rd</sup> parameter.  
The received delimiter char is appended to the string, but the string length is not incremented, so the delimiter disappears, but is still present (invisible). The 3<sup>rd</sup> parameter must be of the type char.



# AVRco Compiler-Manual

If the string is shorter than the byte number specified, the string gets filled to the whole length. The additional bytes are read, but they are not stored. The length byte of the string is set to the number of the read bytes, but limited to the sizeOf(str).

**Function** *Read* (*p* : *Function*; **var** *st* : *string*);  
**Function** *Read* (*p* : *Function*; **var** *st* : *string*; *count* : *byte*);  
**Function** *Read* (*p* : *Function*; **var** *st* : *string*; *limiter* : *char*);

**Read** (*proc* (2), *st1*, 4);            {*st1* is filled}  
**Read** (*proc*, *st2*, 20);            {*st2* is filled with 20 chars}  
**Read** (*proc*, *st2*, #0);            {*st2* is filled until a #0 appears}

The function, which is called, **may** have a parameter. In this case it has to be a constant.

## 4.13.5 READLN

ReadLn reads from a Device until either a CarriageReturn/LineFeed \$0D+\$0A (#13+#10) is found or the supplied string is full.

**Procedure** *ReadLn* (*DeviceFunc* : *function*; **var** *str* : *string*);

*ReadLn* (*SerInp*, *st*);



## 4.14 Error Handling

### 4.14.1 RUNERR

Function. Reading resets the error flag. Many operations set the variable RunErr depending on their result. A RunErr is not absolutely critical, so the program continues run. An error can be an integer-overflow after a multiplication, for example. All operations, which may cause a RunTimeError, set the Flag if necessary. As opposed to the procedure RunTimeError the RunErr can not be switched off and is always active.

The following operations are able to set the flag RunErr:

#### Division

With a zero-division. In this case the highest possible value is returned.

With a Float the highest possible value is returned.

With a Float underflow zero is returned.

#### Multiplication

With an Overflow. Result for example of byte  $\$64 \times \$64 = \$10$

With a Float Overflow the highest possible value is returned.

With Float underflow zero is returned.

#### Addition

With an Overflow. Result for example of byte  $\$84 + \$84 = \$08$

With a Float Overflow the highest possible value is returned.

#### SQR

With a Float Overflow the highest possible value is returned.

#### SQRT

With Float underflow zero is returned. For a negative argument.

#### Type Conversion

With overflow. Float to Word, to Int, to LongWord, to LongInt

#### StrToInt

With a faulty string or Overflow.

#### StrToFloat

With a faulty string.

#### Indexed String and Array Manipulation

With a wrong index

### 4.14.2 RUNTIMEERR

Declaration of the RunTimeError function.

If the switch **RangeCheck** is "on" and the system procedure "RunTimeErr" is imported, so it is checked after every index access (string/array) and its calculation, if the index is valid. If the index is invalid, the procedure is called and the error code 1 is passed within the working register (\_ACCB). The error code 2 specifies an illegal string length-operation.



# AVRco Compiler-Manual

The size and the runtime speed can be optimized by a considered use of compiler switches (+/-). If a compiler switch is still active at the end of the program (end.), then it is also used for the system library.

If the procedure "RunTimeErr" is not imported, the two compiler switches have no meaning and they are ignored. Stringcopy statements (e.g. str: = 'abcd'; ) are always processed without overrun. Here an additional watch is not necessary.

Please also pay attention to **StackSize** and **Compiler Switches**.

*Define StackSize = 32, iData; {32 bytes in iData}*

*procedure RunTimeErr;*

**begin**

*DisableInts;*

*{WatchDog ??}*

*{\_ACCB contains runtime error number}*

*{0 -> software stack or Frame overflow}*

*{1 -> string or array index error}*

*{2 -> string length error}*

*{3 -> reserved}*

*{4 -> convert error float -> ordinal}*

*{5 -> float overflow}*

*{6 -> float underflow}*

**ASM;**

*; store error*

*MOV errnum, \_ACCB;*

*; do not use high level instr if Stack overflow is possible*

*TST \_ACCB;*

*; if \_ACCB = zero then stack error*

*BRNE NOTSTACK;*

*; do something with stack error*

*NOP*

*NOP*

*NOTSTACK:*

**ENDASM;**

**end;**

## **Attention:**

RunTimeErr must not be called from the program itself !!

### **4.14.3 CLEARRUNERR**

Resets a RunTimeError

*Procedure ClearRunErr;*

## 4.15 Multi-Task Functions

The AVRco contains a multitasking-system, which is supported by many functions and procedures. Upto 15 processes and tasks can be defined. These are called periodically by the scheduler, depending on their priority and their state.

Assignments can be done in the background without co-operation with the main program. The processes and the main program are able to communicate with pipes and semaphores.

Tasks are specialized processes, which are called cyclically by the scheduler, whereby the time interval is constant. So tasks can do jobs that have to run in a fixed time grid, for example the PID-controller.

Nearly all of the following functions that expect a process/task name accept also the identifier "SELF". So it is possible for the calling process/task to control itself.

```
Sleep (self, 10); //suspends the actual process for 10 ticks
```

Further most functions can use the process/task ID in place of the name.

```
Sleep (const ProzessID, Ticks: word);
```

ProcessID must be a numerical constant (see below).

### **Attention!**

All suspend and wait functions can only be used if the Idle-process is imported or the application takes care that never all processes including the Main are sleeping, suspended or waiting at the same time.

### 4.15.1 SLEEP

Process Sleep in ticks. The process or task stays inactive for n ticks and is then "woken up" by the scheduler.

```
Procedure Sleep (p : process; t : word);
```

```
Sleep (process1, 50);
```

### 4.15.2 SUSPEND

The process/task is stopped and stays inactive until there is a "resume". Because of that a resume has to be executed from outside the procedure or task.

```
Procedure Suspend (p : process);
```

```
Suspend (process1);
```

### 4.15.3 SUSPEND ALL

```
SuspendAll (Processes, Tasks); // disable processes + tasks  
SuspendAll (Processes); // disable processes only  
SuspendAll (Tasks); // disable tasks only
```

Subsequently the *Idle Process* (if defined) runs. Without *Idle Process* the control switches to the *Main Process*.



# AVRco Compiler-Manual

## 4.15.4 RESUME

A process/task, which is de-activated by "Suspend" or "Sleep" gets activated.

**Procedure** *Resume* (*p* : process);

*Resume* (process1);

## 4.15.5 RESUMEALL

*ResumeAll* (Processes, Tasks); // resume processes + tasks

*ResumeAll* (Processes); // resume processes only

*ResumeAll* (Tasks); // resume tasks only

## 4.15.6 PRIORITY

The importance of the run-time part of a process is specified by the priority. The process processes number of system ticks, which is given by the priority without an interruption. The interval of task calls is specified by the priority in case of tasks. A task runs a maximum of one SysTick without an interruption.

See also chapter *Multitasking Programming – Priority!*

Process min/max Priority = 1..15                      Default=3

Task min/max Priority = 1..255                      Default=5

### **Attention:**

A Task priority of 1 is only for special purposes because this task then is called every SysTick by the Scheduler. Hence no other task must be active at the same time.

**Procedure** *Priority* (*p* : process; *prio* : byte);

*Priority* (process1, 12);

*Priority* (task1, 5);

### 4.15.6.1 GetPriority

returns the actual priority of a process/task

**Function** *GetPriority* (*prcs* : process|task) : byte;

## 4.15.7 MAIN\_PROC

If MultiTasking is imported, the Main Program also runs as a process and it's parameters and functions, like processes, can be manipulated (*lock*, *unlock*, *priority*).

To identify the main use the fixed name **Main\_Proc**

*Priority* (Main\_Proc, 5);                      //Default=5

*Lock* (Main\_Proc);

## 4.15.8 IDLE PROCESS

If all processes including the MAIN\_PROC are stopped by SLEEP, SUSPEND or WAIT there is no process which consumes the CPU time.

There is optional IDLE process implemented and so this problem doesn't exist anymore. All processes can be stopped without making the system unstable. Without the IDLE process at least MAIN must always run and take the role of the Idle process.

Because this IDLE process also must be able to handle interrupts it needs an environment with stack and frame. Furthermore the code/Flash consumption will be increased somewhat and the Scheduler (process handler) is also minimally increased and slowed down a few usec.

The import is done by an enhanced Scheduler Define.

The standard Define (without IDLE process) is:

```
Define Scheduler = DataArea;
```

To import the Idle process the IDLE stack and frame size must be added:

```
Define Scheduler = IdleStack, IdleFrame, DataArea; //use optional idle process
```

The minimum stack and frame size is 10 Bytes.

### 4.15.8.1 On Idle Process

Callback procedure. The Idle process must be defined (with the Scheduler define):

```
Procedure OnIdleProcess;
```

If the application implements the procedure

```
Procedure OnIdleProcess;  
begin  
...  
end;
```

then "OnIdleProcess" is called with each restart of the Idle process.

#### **Attention:**

The Idle process always "lives" only one SysTick (like a Task). So this procedure must not last longer as a SysTick otherwise it will be canceled by the Scheduler.

## 4.15.9 SCHEDULE

The process/task is suspended here and the control is passed to the scheduler. The scheduler now looks for the next process with the highest priority, and this process gets activated.

```
Procedure Schedule;
```

```
Schedule;
```

Within a process or task or also within procedures and functions.

### 4.15.10 SCHEDULER ON/OFF

The Process-scheduler can be stopped with "SchedulerOff" and continued with "SchedulerOn".

If the Scheduler is stopped, only the current process runs. The Scheduler will be skipped completely.

```
Procedure SchedulerOff;
```

```
Procedure SchedulerOn;
```



## 4.15.11 LOCK

The whole run-time is placed at disposal of a process. Interrupts also get processed. Also applies to the Main\_Proc.

*Procedure Lock (p : process);*

*Lock (process1);*  
*Lock (Main\_Proc);*

A *Lock(Process)* disables the scheduling of this process, the scheduler doesn't pass the control to another process but Tasks are not affected. Tasks are treated like interrupts

## 4.15.12 UNLOCK

A locked process is released, enabling the scheduler again.

*Procedure UnLock (p : process);*

*UnLock (process1);*  
*UnLock (Main\_Proc);*

## 4.15.13 RESET PROCESS

This function re-initializes a process completely and suspends it. It can not be applied to the currently running process.

*Procedure ResetProcess (P : Name | i : ID);*

## 4.15.14 SEMAPHORE

### 4.15.14.1 WAITSEMA

A process/task becomes itself inactive until a special semaphore is > 0. When this happens the scheduler makes the task active again. The semaphore is automatically decremented by one.

*Function WaitSema (s : semaphore [; timeout: word]) : boolean;*

*WaitSema (sema1);*

Only within a process or task.

*TimeOut* is optional. If omitted, the process waits until the semaphore is > 0.

The same is true if the *TimeOut* is set to 0000.

If the value is > 0 the wait functions returns after (*TimeOut* \* *SysTicks*). The result of the function is true if there was no timeout.

With tasks *TimeOut* is not possible and is ignored.

*WaitSema* can only be used if the *Idle-process* is imported or the application takes care that never all processes inuding the *Main* are sleeping, suspended or waiting at the same time.

## 4.15.14.2 ProcWaitFlag

The proper use of WaitSema can save a lot of MultiTasking process time. But this function expects a special variable of type Semaphore which then is decremented by one if it is > 0.

If the semaphore must not be changed by the Scheduler or if a variable of any kind must be watched by the Scheduler (var <> 0) a more general function must be used:

**Function** *ProcWaitFlag (Flag : var[; timeout : word]) : boolean;*

Here the variable Flag can be of any type. Unlike WaitSema, this variable will not be changed by the function. The other rules are the same as with WaitSema.

## 4.15.14.3 SETSEMA

The content of the semaphore is set to the required value.

**Procedure** *SetSema (sema : semaphore; v : byte);*

## 4.15.14.4 INCSEMA

A semaphore is incremented by one.

**Procedure** *IncSema (s : semaphore);*

*IncSema (sema1);*

## 4.15.14.5 DECSEMA

A semaphore is decremented by one. If the decrement was successful (Sema was > 0) a true is returned, otherwise a false.

**Function** *DecSema (s : semaphore) : boolean;*

*DecSema (sema1);*

## 4.15.14.6 SEMASTAT

Returns the content of a semaphore.

**Function** *SemaStat (s : semaphore) : byte;*

*b:= SemaStat (sema1);*

## 4.15.15 PIPES

### 4.15.15.1 WaitPipe

A process/task makes itself inactive until a special pipe has data. When this happens the scheduler makes the task/process active again.

**Function** *WaitPipe* (*p* : pipe [*;* timeout: word]) : boolean; {also RxBuffer and RxBuffer1, -2, -3}

*WaitPipe* (*pipe1*);  
*WaitPipe* (*RxBuffer*);

Only within a process or task.

*TimeOut* is optional. If omitted, the process waits until the semaphore is > 0.

The same is true if the *TimeOut* is set to 0.

If the value is > 0 the wait functions returns after (*TimeOut* \* *SysTicks*). The result of the function is true if there was no timeout. With tasks the *TimeOut* is not possible and is ignored.

*WaitPipe* can only be used if the *Idle*-process is imported or the application takes care that never all processes inuding the *Main* are sleeping, suspended or waiting at the same time.

### 4.15.15.2 PipeFlush

Empties a pipe completely.

**Procedure** *PipeFlush* (*p* : pipe); {also RxBuffer and RxBuffer1, -2, -3}

*PipeFlush* (*pipe1*);  
*PipeFlush* (*RxBuffer*);

### 4.15.15.3 PipeSend

Inserts an argument into a pipe. The result shows whether the operation was successful. If the pipe is full, a false will be returned.

**Function** *PipeSend* (*p* : pipe; *v* : type) : boolean;

*bo* := *PipeSend* (*pipe1*, *value*);

### 4.15.15.4 PipeRecv

Fetches an argument from a pipe. The function only returns if the operation was successful. In order to waste no run-time, processes should use "PipeStat" or better still "WaitPipe".

**Function** *PipeRecv* (*p* : pipe) : type;

*val* := *PipeRecv* (*pipe1*);

### 4.15.15.5 PipeStat

The count of the number of parameters is interrogated.

**Function** *PipeStat* (*p* : pipe) : byte;

*b* := *PipeStat* (*pipe1*);

The function *PipeStat* can also be used with *RxBuffer*, *RxBuffer1*, -2, -3 and *TxBuffer*, *TxBuffer1*, -2, -3 of the serial interfaces.



## 4.15.15.6 PipeFull

The full-status of a pipe is interrogated.

**Function** *PipeFull* (*p* : *pipe*) : *boolean*;

*bo* := *PipeFull* (*pipe1*);

## 4.15.16 PROCESS ID

Some functions or procedures which are used by several processes or tasks have to behave differently, depending on the calling process of task. Such a procedure must be able to establish which process/ task is calling it. To this end following three functions are used.

The main program (Main\_PROC) always has the process-ID 0 and has the process name 'Main\_Proc'.

### 4.15.16.1 ISCURPROCESS

Function. Interrogation if the actual process has the ID x. or the Name ,abc'.

**Function** *isCurProcess* (*ID* : *byte*[:; *Name* : *ProcName*]) : *boolean*;

*bo* := *isCurProcess* (0); { 0 = Main }

*bo* := *isCurProcess* (Main\_Proc);

### 4.15.16.2 GETCURPROCESS

The process ID of the actual process/task is interrogated.

**Function** *GetCurProcess* : *byte*;

*Id* := *GetCurProcess*;

### 4.15.16.3 GETPROCESSID

The process ID of a process is interrogated.

**Function** *GetProcessID* (*ProcName*) : *byte*;

*Id* := *GetProcessID* (Main\_Proc);

## 4.15.17 PROCESS STATE

The state of a process or task can be found.

To this end the system exports the enumeration:

*tProcessState* = (*eProcStop*, *eProcRun*, *eProcidle*, *eProcWait*, *eProcSleep*, *eProcLock*);

and the function:

**Function** *GetProcessState* (*name* : *process|task*) : *tProcessState*;

```
if GetProcessState (ProcessA) = eProcSleep then  
  ...  
endif;
```

```
case GetProcessState (Main_Proc) of  
  eProcStop :  
    inc(bb); |  
  eProcRun :  
    inc(bb); |  
  eProclidle :  
    inc(bb); |  
  eProcWait :  
    inc(bb); |  
  eProcSleep:  
    inc(bb); |  
  eProcLock :  
    inc(bb); |  
endcase;
```

## 4.15.18 DEVICE LOCK

When using device drivers in a MultiTasking environment sometimes it is necessary to lock these drivers against other processes/tasks. To this end the type "*DeviceLock*" was implemented (only for global Vars and not for Arrays or Records).

Processes which access the Device should use this mechanism to signal other processes that this driver is in use momentarily.

### 4.15.18.1 SetDeviceLock

**Function** *SetDeviceLock* (*d* : *DeviceLock*) : *boolean*;

Returns with a true if the Device is free and changes the value into locked.

If the Device is in use the function returns a false.

With a true the process now can use the Device driver. When the job is done it should free the Device with the function *ClearDeviceLock*.

### 4.15.18.2 ClearDeviceLock

**Function** *ClearDeviceLock* (*d* : *DeviceLock*) : *boolean*;

Free the Device. If the Device was already released the function returns a "*false*", otherwise it returns a "*true*".

### 4.15.18.3 TestDeviceLock

The function checks a device lock state without changing it. It returns a "*true*" if the Device is free. Otherwise it returns a "*false*".

**Function** *TestDeviceLock* (*d* : *DeviceLock*) : *boolean*;

## 4.15.18.4 WaitDeviceFree

**Function** *WaitDeviceFree* (*s* : *DeviceLock* [*; timeout: word*]) : *boolean*;

With this function the process will be suspended until the Device is released.

*TimeOut* is optional. If omitted, the process waits until the device is free.

The same is true if the *TimeOut* is set to 0000.

If the value is > 0 the wait functions returns after (*TimeOut* \* *SysTicks*). The result of the function is true if there was no timeout. If a timeout occurred the "DeviceLock" is automatically set to *locked*.

With tasks the *TimeOut* is not possible and is ignored.

*WaitDeviceFree* can only be used if the *Idle*-process is imported or the application takes care that never all processes inuding the *Main* are sleeping, suspended or waiting at the same time.

```
var DevSema : DeviceLock;
```

```
Procedure Init;
```

```
begin
```

```
  ClearDeviceLock(DevSema);           // first free of the device
```

```
  ...
```

```
end;
```

```
Process CheckDevice(32,64 : idata);
```

```
begin
```

```
  WaitDeviceFree(DevSema);
```

```
  // enter the device driver
```

```
  ....
```

```
  // free the device driver
```

```
  ClearDeviceLock(DevSema);
```

```
  ...
```

```
end;
```

## 4.15.19 Stack and Frame Usage

### 4.15.19.1 GETSTACKFREE

Get the Stack usage of tasks and processes at runtime.

The function returns a word with the minimum count of unused bytes in the Stack so far.

**Function** *GetStackFree* (*p* : *Process|Task*) : *word*;

```
ww:= GetStackFree (Main_Proc);
```

```
ww:= GetStackFree (Task1);
```

```
ww:= GetStackFree (Proc1);
```

### 4.15.19.2 GETFRAMEFREE

Get the Frame usage of tasks and processes at runtime.

The function returns a word with the minimum count of unused bytes in the Frame so far.

**Function** *GetFrameFree* (*p* : *Process|Task*) : *word*;

```
ww:= GetFrameFree (Main_Proc);
```

```
ww:= GetFrameFree (Task1);
```

```
ww:= GetFrameFree (Proc1);
```



# AVRco Compiler-Manual

## 4.15.19.3 CHECKSTACKVALID

Checks a stack overflow. When an overflow happens the result is \$FFFF. Without an overflow the result is the free stack size.

**Function** *CheckStackValid* (*p* : *Process|Task*) : *integer*;

## 4.15.19.4 CHECKFRAMEVALID

Checks a frame overflow. When an overflow happens the result is \$FFFF. Without an overflow the result is the free frame size.

**Function** *CheckFrameValid* (*p* : *Process|Task*) : *integer*;

Both functions need the System Import *StackChecks*

If MultiTasking is not implemented so no argument (Process/Task) must be supplied, otherwise with the main program part the argument *Main\_Proc* must be used.

## 4.15.20 SCHEDULER CALL BACK

For debugging it is sometimes necessary to know when the Scheduler needs to switch from one process/task to another.

Because this time depends heavily on the program implementation (count of processes/tasks, priorities etc.) this can only be evaluated at runtime. To help with this there are two predefined procedures:

**Procedure** *OnSchedulerEntry*;

**Procedure** *OnSchedulerExit*;

If the application implements these procedures:

**Procedure** *OnSchedulerEntry*;

**begin**

...

**end;**

**Procedure** *OnSchedulerExit*;

**begin**

...

**end;**

"OnSchedulerEntry" is called every time at the entry into the Scheduler and "OnSchedulerExit" is called every time on exit of the Scheduler.

For example with the simulator it's possible to find out the time the Scheduler used. To do this a breakpoint must be placed on each of these 2 procedures. The heavily varying results depend on the current job the scheduler is doing, e.g. task/process-switch yes/no etc. The resulting time cycle count also represents the total time of the disabled global interrupt whilst scheduling.

### **Important:**

1. There must be no local parameters.
2. If Pascal statements or registers are used, the related registers must be saved before their usage and restored afterwards.
3. Operations which change the CPU flags can only be executed after the status register of the CPU has been saved.

## 4.16 PID-Controller

### Pseudo-Record

A PID-controller is often used in technical applications, for example temperature regulation, servos, rpm-controllers etc.

A PID-controller has to have entry-parameters: the nominal value = 'Required' and the *actual* value. Four parameters, which are normally only adjusted once, are *pFactor*, *iFactor*, *dFactor* and *sFactor*. The controller-output, which goes to the actuator (heating, motor etc.) is calculated by the function 'Execute'

The controller type is determined by the two initializing parameters '*iLimit*' and '*dIntVal*'.

### **iLimit**

is of the type LongWord (0..100000) and determines the max. size of the **I**-part (clipping). If *iLimit* = 0, the **Integral**-value of the controller is not calculated and is omitted (e.g. PD-controller).

### **dIntVal**

is of the type Byte (0, 1, 2, 4, 8, 16, 32) and determines the degree step of the calculation of the **D**-part (gradient). If *dIntVal* = 0, the **Differential**-value of the controller is not calculated and so is omitted (e.g. PI-controller). If the value = 1, the gradient from the last nominal value to the actual nominal value is calculated. In the remaining cases a corresponding array is framed, which absorbs the history of the last *n* nominal values. So the gradient can be calculated over a large number of nominal values.

The controller calculates internally with **LongInteger**. So overflows within Execute are not expected.

### Imports

As always with AVRco the driver must be imported.

```
Import SysTick, Pids, ...;
```

```
Var Pid1 : PIDcontrol[iLimit, dIntVal];
```

```
{Init}
```

```
Pid1.pFactor:= 1000;
```

```
Pid1.iFactor:= 2500;
```

```
Pid1.dFactor:= 678;
```

```
Pid1.sFactor:= 10000;
```

```
{Run}
```

```
Pid1.Actual:= 500;
```

```
Pid1.Nominal:= 550;
```

```
PWM1:= Pid1.Execute;
```

### 4.16.1 pFACTOR

```
PIDname.pFactor:= p;
```

Charging-factor of the P-value. The Pvalue is the difference between *nominal* and *actual* value, also known as error. The execute operation calculates the internal pValue. *PIDname.pValue*

### 4.16.2 iFACTOR

```
PIDname.iFactor:= i;
```

Charging-factor of the I-value. The I-value is the sum of the differences between *nominal* and *actual* value. In general: integral- value of the errors. The execute operation calculates the internal iValue. *PIDname.iValue*



# AVRco Compiler-Manual

## 4.16.3 dFACTOR

*PIDname.dFactor:= d;*

Charging-factor of the D-value. The D-value is the gradient of the differences between the *nominal* and *actual* value. In general: error gradient.

The execute operation calculates the internal dValue. *PIDname.dValue*

## 4.16.4 sFACTOR

*PIDname.sFactor:= s;*

Charging-factor of the output-value (result). The output-value is the sum of P, I and D.

In general:  $((P \times pFactor) + (I \times iFactor) + (D \times dFactor)) \text{ div } sFactor$ .

## 4.16.5 NOMINAL

*PIDname.Nominal:= i;*

Nominal value. Calculates, together with the *actual*, the deviation, which is incorporated in the P, I and D values.

## 4.16.6 ACTUAL

*PIDname.Actual:= i;*

Actual-value. Returns, together with the *nominal*, the deviations, which is incorporated in the P, I and D values.

## 4.16.7 EXECUTE

*i:= PIDname.Execute;*

Function, which is calculating the output value by using the given values (nominal, actual and factors). The result is an integer value.

## 4.17 Functions depending on HardWare

### 4.17.1 PROCCLOCK

Processor Clock in Hertz. Is needed for Software Delays, for example *Mdelay* and *uDelay*, as well as for the calculation of the *SysTick*. It **has to be** defined.

```
Define ProcClock = 4000000;    {4Mhz clock}
```

### 4.17.2 STACKSIZE, RAMpage

The required stack size (Software Stack) **has to be** defined. It is needed for the allocation and testing of stack- and program inter-variables. Because the stack grows and shrinks, from “above” to “below” during the program execution and the used variables are established from “below” to “above” by the programmer, the compiler is able to give a warning, if the stack could “grow into” the variables.

A stack size has to be defined to help the compiler, because the actual size of the stack can only be established during the run-time. With an overlap of program variables and stack (+StackSize) the compiler gives a warning.

The lowest StackSize is 16. The required Ram-area of the stack also has to be specified (iData, xData).

```
Define StackSize = 32, iData;    {32 bytes in iData}
```

### 4.17.3 FRAMESIZE, RAMpage

The required FrameSize (passing parameter and local variable) **has to be** defined. It is needed for allocation and testing of passing parameters (procedures, functions) and local variables.

Because the frame grows and shrinks, from “above” to “below” during the program execution and the used variables are established from “below” to “above” by the programmer, the compiler is able to give a warning, if the frame could “grow into” the variables.

A frame size has to be defined to help the compiler, because the actual size of the stack can only be established during the run-time. With an overlap of program variables and frame (+FrameSize) the compiler gives a warning.

The lowest FrameSize is 8. The required Ram-area for the frame also has to be specified (iData, xData).

```
Define FrameSize = 32, iData;    {32 Bytes in iData}
```

## 4.17.4 TASKSTACK, RAMpage

The required stack size (all tasks have the same stack-area) **has to be** defined, if tasks have been imported. Is needed for the allocation of the memory.

The lowest value of TaskStack is 8. The required Ram-area of the stack also has to be specified (iData, xData).

```
Define TaskStack = 32, iData;      {32 bytes in iData}
```

## 4.17.5 TASKFRAME

The required frame size (all tasks have the same stack-area) **has to be** defined, if tasks have been imported. Is needed for the allocation of the memory.

The lowest value of TaskFrame is 8. The required Ram-area of the frame also has to be specified (iData, xData).

```
Define TaskFrame = 16;           {16 Byte}
```

## 4.17.6 SCHEDULER

The required memory page (iData, xData) for the process- and task administration **has to be** defined, if processes or tasks have been imported.

The scheduler does the task- and process administration and switches them over. The scheduler is called by the timer0 action of the SysTick. Because it is an interrupt, the global interrupt stays blocked until the scheduler has done its job. This time can be, depending on the jobs of the SysTick, number of processes etc., upto 500usec. If there is the danger that, for example, fast interrupts can not be interrogated, because the global interrupt is still blocked, the global interrupt within the timer0 can be immediately set free by using the additional instruction "interruptible".

```
Define Scheduler = iData;        {Scheduler in iData}  
Define Scheduler = iData, interruptible; {Scheduler in iData}
```

## 4.17.7 SYSTICK

Timer-controlled interrupt for time functions. If SysTick has been imported by the *IMPORT* clause, the required tick time has to be specified. The value can be in the range of 0.1..100 (msec). This value can be expressed in floating point, which allows a precise adjustment for the needs of the application. SysTick is implemented as a hardware interrupt of a timer. Within the AVR normally this is the timer0 (8bit timer). If the selected CPU has the Timer2 (also an 8bit timer) it also can be used for the SysTick. The used timer is at then not useful for the program itself. Manipulations on the timer hardware or register can lead to a crash of the program.

A good value for the tick is 10 (msec). So the interrupt does not make the system *dense* and the jobs of the ticks, for example debouncing of the *SwitchPort* can be settled well. With the AVR *SysTick Timer0 (or Timer2)* is the only interrupt, which is directly used and evaluated by the system. The programmer has to initialize hardware interrupts himself from with the program. These interrupts have to be completely processed in the pre-defined procedure *Interrupt xxx;*

The Timer is loaded from the variable "*SysTickTime*". The application is able to manipulate the SysTick time within certain limits at runtime by changing this variable.



SysTick is absolutely necessary for many Drivers, so normally **SysTick** always **has to be imported and defined**.

```
Import SysTick;
Define ProcClock = 4000000;    {4Mhz clock }
        SysTick   = 10;        {10msec Tick}
or
        SysTick   = 5.5;       {5.5msec Tick}
or
        SysTick   = 8.0, Timer2; {8msec Tick}
```

#### 4.17.7.1 OnSysTick

For timing jobs which must be executed through SysTick the procedure "OnSysTick" is implemented. Because this is a call-back out of a Timer interrupt the global interrupt is disabled here. Be carefull and don't overload this function in order to avoid long interrupt disable time. "OnSysTick" is called with each SysTick (Timer Interrupt).

```
Procedure OnSysTick;
Procedure OnSysTick (SaveAllRegs);
```

Implementation in the application:

```
Procedure OnSysTick;
begin
...
end;
```

or

```
procedure OnSysTick (SaveAllRegs); // not possible if MultiTasking is used!
begin
...
end;
```

#### **Attention:**

With MultiTasking import the "SaveAllRegs" option must not be used. It's not applicable here and only the standard registers ACCA, ACCB, ACCLO, ACCHI are preserved.  
With non-MultiTasking and the option "SaveAllRegs" there are no limits, but remember that saving and restoring all regs takes time

With MultiTasking or non-MultiTasking without the option "SaveAllRegs" the following is true:

1. Local parameters are prohibited.
2. The register SREG, \_ACCA/R17, \_ACCB/R16, \_ACCCLO/R30 and \_ACCCHI/R31 are saved always.  
If Pascal statements or other additional registers are used, then these register must be saved manually before their usage and restored afterwards.

#### 4.17.7.2 SysTickStop

For special purposes procedure for SysTick manipulation

```
Procedure SysTickStop;           // Disable Timer Interrupt, STOP Timer
```

This function must always be used in conjunction with *SysTickStart*.

#### 4.17.7.3 SysTickStart

For special purposes procedure for SysTick manipulation

```
Procedure SysTickStart;         // Start Timer Interrupt, preset TCNT, Start Timer
```

This function must always be used in conjunction with *SysTickStop*.



# AVRco Compiler-Manual

## 4.17.7.4 SysTickRestart

Re-initializes the hardware timer which triggers the SysTick.

## 4.17.7.5 SysTickDisable

For special purposes procedure for SysTick manipulation

*Procedure SysTickDisable; // disable only Timer Interrupt, Timer keeps running*

This function must always be used in conjunction with *SysTickEnable*.

## 4.17.7.6 SysTickEnable

For special purposes procedure for SysTick manipulation

*Procedure SysTickEnable; // enable only Timer*

This function must always be used in conjunction with *SysTickDisable*.

## 4.17.7.7 SystemTime (\*4\*)

Sometimes it is necessary to have a system time. This is not a clock but the count of SysTicks elapsed since Startup/Reset. For this the the SysTick must be imported and also an import of the SystemTime:

*From SysTick Import SystemTime16; // 16bit word system time*

or

*From SysTick Import SystemTime32; // 32bit longword system time*

This import defines and exports a variable, either as a WORD or LONGWORD. This variable becomes incremented with every SysTick and can be read or written by the application. Because of the activated attribute **locked** each access by the application is executed with disabled interrupts!

*Var SystemTime16 : word; locked; // 16bit word system time*

or

*Var SystemTime32: longword; locked; // 32bit longword system time*

## 4.17.8 ENABLEINTS

Enables the global interrupt. Often it is necessary, that the main program is able to do various initializations (ports etc) before the system tick starts to run. So the global interrupt is not enabled within the system itself, even if interrupts have been imported/defined (SysTick, Serial etc). The application program can and **has to** call the system procedure "EnableInts" once, that interrupts are possible.

*{Program Body = Main}*

**begin**

*...*

*EnableInts:*

**Loop**

*...*

**EndLoop;**

**End.**

## 4.17.9 START\_PROCESSES

Starts processes and tasks. Resets the timer (SysTick) and sets the global interrupt free. This procedure **replaces** the procedure **EnableInts** if processes or tasks are imported.

It is often necessary for the main program to make several initializations (ports etc.) before the system tick/processes/tasks start. Therefore the system itself does not enable the global interrupt. Even when processes are imported/defined (SysTick, Serial etc.). The application **has to call** the system-procedure "Start\_Processes" once to enable Interrupts.

```
{Program Body = Main}  
begin  
...  
  Start_Processes:  
  Loop  
...  
  EndLoop;  
End.
```

## 4.17.10 DISABLEINTS

Disables the global interrupt. Now interrupts are no longer accepted. Interrupts should not be disabled for too long. Enable the interrupts by EnableInts.

*Procedure DisableInts;*

## 4.17.11 NOINTS, RESTOREINTS

EnableInts and DisableInts have an additional function, they set/reset a system flag, so that the system always knows which interrupt state the application has. If the system must disable the interrupt it can only re-enable it if the interrupt was enabled before the system had disabled it. Similar is true for parts of the application where the actual interrupt state is unknown.

This handling is supported by NoInts and RestoreInts. NoInts disables the interrupt regardless of its actual state but does not change the system flag. RestoreInts then re-enables the interrupt only if the system flag is set. This means that the interrupt was enabled before NoInts was called.

Use these functions very carefully!!

## 4.17.12 CPUSLEEP

The CPU goes to sleep, i.e. the program is stopped and internal activities of the CPU are switched off. The CPU awakes only with an external interrupt or a reset. The parameter is written into the corresp. Register of the CPU (mcucr) and defines the kind of the sleep mode.

*Procedure CpuSleep (sleepcmd : byte);*

*CPUsleep (MCUCR or \$30);                    // absolute powerdown*

### Sample Program:

A sample program can be found in the directory **..E-Lab\AVRco\Demos\Sleep**. It has three separate programs that demonstrate the Sleep Modes of the AVR in a simple way.

## 4.17.13 POWERSAVE

The CPU sleeps n SysTicks. SysTick must be imported. The parameter "mode" is stored into the control register of the CPU (see also CPUSleep), the parameter "ticks" defines the time of the PowerDown, counted in SysTick cycles. The parameter "mode" must provide that at least the timer which is used by SysTick runs and can generate interrupts.

The SysTick Timer wakes the CPU with each tick. The parameter "Ticks" is decreased by one and if not zero, the CPU again falls into the PowerDown Mode.

*Procedure PowerSave (const mode : byte; const ticks : word);*

## 4.17.14 WATCHDOG

Imports with an "Import" the WatchDog. With "Define" the prescaler of the timer is adjusted, if it is available.

*Import SysTick, WatchDog;*

```
Define SysTick = 4000000;      { 4MHz }  
        WatchDog = 7;         { WatchDog Prescaler}
```

Possible is also a better readable definition:

```
Define WatchDog = msec16;     // msec32, msec64, msec125, msec250, msec1000, msec2000
```

## 4.17.15 WATCHDOGSTART

Initializing and start of the WatchDog, if it is available .

*Procedure WatchDogStart;*

## 4.17.16 WATCHDOGSTOP

Stops the WatchDog, if it is available.

*Procedure WatchDogStop;*

## 4.17.17 WATCHDOGTRIG

Triggers the Hardware WatchDog of the CPU. The general enabling of the Watchdog depends on the CPU and has to be adjusted with device or import/define if necessary.

*Procedure WatchDogTrig;*

## 4.17.18 GETWATCHDOGFLAG

Function. Returns a byte which is the copy of the "MCUSR" register content after the last reset or PowerOn. So the application can determine what causes the last RESET by examining this byte result. The content and it's bits can have different meaning with different CPUs

*Function GetWatchDogFlag : byte;*

**Note:** The previous function "*WatchDogFlag*" is replaced by the new "*GetWatchDogFlag*".

## 4.17.19 {\$NOWATCHDOGAUTO}

Compiler switch. Only for special purpose. If active no automatic watchdog retriggers are done in the delays like mDelay etc. and in system caused delays.

## 4.17.20 ENABLE\_JTAGPORT

In order to use JTAG port of a AVR at runtime as a standard IO-port the application must be able to disable and enable the JTAG Port.

*Procedure Enable\_JTAGport;*

## 4.17.21 DISABLE\_JTAGPORT

In order to use JTAG port of a AVR at runtime as a standard IO-port the application must be able to disable and enable the JTAG Port.

*Procedure Disable\_JTAGport;*

## 4.18 EEPROM

Some of the older single chips types have an internal EEPROM memory. Most of the newer versions generally have an EEPROM.

The problem is that usually access is done with an index- or address-register and additional control registers. This contrasts to the normal addressing within a compiler. At the very least the writing of the EEPROM needs additional outlay. An access-implementation in a direct high-level-language by the application program is possible, but it makes no sense, because the generated code would increase. An Inline-Assembler solution is imaginable, but it is not everyone's taste and also often faulty.

If the chosen processor has an EEPROM and is this EEPROM registered in the Description-File (xxx.dsc), the AVRco offers three possibilities for EEPROM-access:

### 4.18.1 Structured Constant

With the compiler switch `{$EEPROM}` and the instruction **StructConst** the following constants are defined as positioned within the EEPROM. An access to the variables into the EEPROM then succeeds automatically.

An indirect addressing by a pointer is possible with *EEPromPtr()*.

This method is efficient and very clear and readable, because it can be inserted into the source without any problems. The disadvantage is, that every single byte, which should be accessed, also has to be defined as a constant.

A separate hex-file is generated, which contains the defined constant, and which can be programmed into the CPU by a programmer. See also **StructConst**.

#### **Important:**

After the definition of the EEPROM StructConst switch back to normal memory with `{$DATA}` or `{$IDATA}`.

### 4.18.2 Variable

With the compiler switch `{$EEPROM}` and the instruction **VAR** the following variables are defined as located within the EEPROM. An access to the variables into the EEPROM then succeeds automatically.

An indirect addressing by a pointer is possible with *EEPromPtr()*.

This method is efficient and very clear and readable, because it can be inserted into the source without any problems. The disadvantage that every single byte which should be accessed also has to be defined as a variable.

#### **Important:**

After the definition of the EEPROM vars switch back to normal memory with `{$DATA}` or `{$:Data}`.

## 4.18.3 Memory Block

The EEPROM is treated as an one-dimensional array of byte. For this an **EEPROM** is pre-defined within the system. With this array there is an access by indexes. But array copy is not possible.

Because the index is a constant or a variable, it is comfortably worked with it. The range check, if it is activated, is also valid. The size of the EEPROM array can be established during the run-time by `SizeOf(EEPROM)`.

The disadvantage of this method is, that an index is not that meaningful as a name, and it has to be administrated.

### **Basic EEPROM Limitations:**

Some Read-Modify-Write operations are not possible.

## 4.18.4 EEPROM Access

All three methods have their own advantages for using the EEPROM. A combination of the methods offers a optimal usage. With StartUp for example, data are read out of the array into available RAM-variables by a pointer within a loop and among other things they are written back later. Within the "normal execution" the EEPROM-variable may be used. Look out for unwanted overlapping!

All CPU's, which have an internal EEPROM, assume that during a writing-access to the EEPROM the interrupts are blocked. The compiler addresses this issue.

While the internal byte-programming-logic is active there can be no further access to the EEPROM. Before there is an access to the EEPROM, the state is interrogated, so it is secured. The writing- and reading-routine waits at the beginning for the release of the EEPROM.

```
{$DATA}
```

```
var
```

```
  b1  : byte;
  w1  : word;
  i1  : integer;
```

```
{$EEPROM}
```

```
var
```

```
  be      : byte;
  we      : word;
  ie      : integer;
  el[@we] : byte;      // low byte
  eh[@we+1] : byte;    // high byte
```

```
{$DATA}
```

```
var
```

```
  b1  : byte;
  w1  : word;
  i1  : integer;
```

```
b1:= be;           {copy out of eeprom to b1}
W1:= we;          {copy out of eeprom we to w1}
i1:= ie;          {copy out of eeprom ie to i1}
```

## 4.19 HEAP (\*P\*)

With the standard version of AVRco and also with many other compilers for embedded applications the memory usage/management is limited to global variables, stack and frame. This means that the memory partitioning is already known at compile-time and not dynamically changeable. This is not a problem in most cases.

In AVRco the memory is allocated from address 0 (register) over the I/O-area upto the iData (or xData) area upward. At the end of the used area there are the stacks and frames. The following memory, if any, is normally not accessible for the program. If there is a need for additional memory at runtime, e.g. for linked lists etc., there is no possibility to provide and use this memory.

With larger systems, e.g. on the PC, the free unused memory can be required and allocated through the heap management. The Profi-version of AVRco there is such a heap management implemented. Dependent on which memory area is defined as the heap (iData, xData), the unused free memory in this area is completely administrated by the heap system.

A special requirement of the heap is that accesses must be done with pointers. Therefore free memory must be requested by the program. The address of the assigned block is returned in a pointer from the heap manager. After the use by the program the memory should be deallocated respectively passed back to the heap manager. Otherwise the memory stays allocated and can't be used from other parts of the program.

It's very important that the address (pointer) at deallocation is the same as with the allocation. Otherwise the deallocation fails and the heap will eventually become is completely allocated and each memory request will fail.

### 4.19.1 Implementation

#### Imports

The driver must be imported as usual with AVRco.

*Import* `SysTick, Heap, ...;`

#### Defines

The memory area which should be used by the heap manager:

```
Define ProcClock = 8000000;    {Hertz}
          SysTick   = 10;       {msec}
          StackSize = $0030, iData;
          FrameSize = $0030, iData;
          Heap      = iData;
```

#### 4.19.1.1 Functions

Memory allocation and deallocation through the heap management functions:

**Function** `GetMem (var ptr : pointer [; const size : word]) : boolean;`

Requests memory from heap with the optional value "size". If size is not passed, there will be a block of memory allocated which the structure needs, where the "ptr" points to. If ptr is a pointer to string[10], so 11 Bytes are allocated. The variable "ptr" can be a pointer of any type.



The function returns a true, if there was sufficient memory for the requested block. The variable ptr returns the address of this memory block. If the function fails, a false is returned and the variable ptr returns the value NIL.

## **Attention:**

The optional parameter “size” should only be used with care and is only intended for special cases.

### **Function FreeMem (var ptr : pointer) : boolean;**

Deallocates used memory. The size block of memory deallocated is as large as the structure needs, where the “ptr” points to. If ptr is a pointer to string[10], so 11 Bytes are deallocated. The variable ptr should be the same which was used when this block was requested by GetMem.

If the memory could be deallocated, this means the above conditions are fulfilled, a true is returned, otherwise a false. A false should **never** occur with correct programming. The variable ptr is always returned with NIL.

### **Function GetMemAvail: word;**

Returns the total amount of free memory. With repeatedly allocation and deallocation it's possible the free memory is partitioned in several blocks.

The returned value doesn't say anything about whether it's available in one piece or not.

### **Function GetLargestBlock: word;**

Returns the value of the largest one-piece unfragmented memory block.

## **Remarks:**

Pointers in general and with the heap in particular are dangerous (Hello C). But if one keeps in mind a few rules there can be new and streamlined programming possibilities (linked list for example).

1. **Always** use **typed** pointers for GetMem and FreeMem.
2. **Always** use the **same** pointer variable for GetMem and the corresponding FreeMem.
3. **Avoid** the usage of the parameter **size**. It's unnecessary with typed pointers.
4. **Avoid manipulating** the pointer itself.
5. Unused memory should be **deallocated** at **soon** as possible, so it can be used again.
6. **Observe** the boolean **result** of GetMem. If **false** there is simply no more memory.
7. If there is a **false** result with FreeMem, you have programmed a big **bug**.
8. Note that with each allocation additional 4 bytes are required for administration purpose.  
So a request for one byte (with Pointer to Byte) a total of 5 bytes are used.
9. Always use the **Simulator** to test and debug your program.
10. If you think it's necessary to use the parameter **size**, define the amount of used bytes of a type or structure with the system function **sizeof()**

**In general, be careful with pointers ☺**



# AVRco Compiler-Manual

## 4.19.1.2 Example

**Program** AVR\_Heap;

**Device** = 90S8515, VCC=5;

**Import** SysTick, Heap;

**From System Import** ;

### Define

```
ProcClock = 6000000;    {Hertz}
SysTick   = 10;        {msec}
Heap      = iData;
StackSize = $0020, iData;
FrameSize = $0010, iData;
```

### Implementation

```
{ $IDATA }
{-----}
{ Type Declarations }
type
tStr10 = string[10];
tArr   = array[0..23] of byte;
tRec   = record
        rb : byte;
        rw : word;
        end;

tpStr  = pointer to tStr10;
tpArr  = pointer to tArr;
tpRec  = pointer to tRec;

{-----}
{ var Declarations }
{ $IDATA }
var
ww      : word;
ptr, ptr1 : pointer;
pStr    : tpStr;
pArr    : tpArr;
pRec    : tpRec;
bool    : boolean;
```

```
{-----}  
{ Main Program }  
{ $IDATA }
```

## **begin**

```
ww:= GetMemAvail;  
ww:= GetLargestBlock;  
bool:= GetMem (ptr, 1);  
ww:= GetMemAvail;  
ww:= GetLargestBlock;  
bool:= GetMem (pStr);  
ww:= GetMemAvail;  
ww:= GetLargestBlock;  
bool:= GetMem (pArr);  
ww:= GetMemAvail;  
ww:= GetLargestBlock;  
bool:= GetMem (pRec);  
ww:= GetMemAvail;  
ww:= GetLargestBlock;
```

```
bool:= FreeMem (ptr);  
ww:= GetMemAvail;  
ww:= GetLargestBlock;  
bool:= FreeMem (pStr);  
ww:= GetMemAvail;  
ww:= GetLargestBlock;  
bool:= FreeMem (pArr);  
ww:= GetMemAvail;  
ww:= GetLargestBlock;  
bool:= FreeMem (pRec);  
ww:= GetMemAvail;  
ww:= GetLargestBlock;
```

```
EnableInts;
```

```
loop
```

```
  Nop;
```

```
endloop;
```

```
end AVR_Heap.
```

## **Sample Program:**

A complete example can be found in the directory `..\E-Lab\AVRco\Demos\Heap`

## 4.20 BOOT VECTORS

The majority of the mega-AVRs support a second vector table in the Boot area. With this it is possible to use interrupts also while running updates with the FlashLoader. The UART is an example.

The problem with the Self-Update of an AVR through the Boot Loader is that at least the original Interrupt Vector Table must also be rewritten. So it is erased first before being rewritten. Then at least for a short period of time no interrupts can be executed. Consequently interrupt driven boot loaders cannot be used.

In some cases it makes sense or it is necessary to work with interrupts in the loader. Most of the Mega AVR support this by a remapping of the vector table into the start of the boot area. This must be done by the application by manipulating the IVSEL bit in the MCUCR or GICR register.

Interrupts vector read accesses then go either into the vector table at address 0 (IVSEL = 0) or into the second vector table at the beginning of the boot area (IVSEL = 1).

The AVRco system supports this, and there is another advantage that the interrupt service routines which reside in the boot area can also be used by the application. In this case the concerning interrupt (vector) is placed into both tables. The related interrupt service routine then must always be implemented in the Boot area.

There are three types of interrupts:

1. The service routine is implemented outside of the boot area. This is the standard case and there is no influence on the boot area and its vector table.
2. The service routine resides in the boot area and has a standard name. Both vector tables have an entry (vector) of this routine. So both the application and the boot loader can use this interrupt.
3. The service routine is implemented in the boot area and has a name extension ***BOOT***, ***INT0\_BOOT*** for example. Then this routine is only useable by the loader and also only this table contains a corresponding entry. The application then can define its own ***INT0*** interrupt.

### **Important**

The BootBlock must start always with this statement:

```
{ $PHASE BootBlock nnnnn }
```

Then a procedure must follow which serves as the entry point (quasi Main) for the BootBlock. Then Interrupt procedures or other functions/procedures can follow. The last line must always be:

```
{ $DEPHASE BootBlock }
```

### **Limitation**

If the system uses an interrupt exclusively, as SerPort does with RxBuffer, then the RxRDY interrupt is not shareable. Each table must have its own entry.

## 4.20.1 Implementation

### Imports

The boot interrupt support must be imported.

*Import* `SysTick, TickTimer, ..;`

*From System Import* `BootVectors, ...;`

## 4.20.2 Functions

*Procedure* `SetVectTabBoot (boot : boolean);`

This procedure should be used for the time critical switch of the vector tables (IVSEL, MCUCR/GICR).

*Procedure* `Boot_Init;`

Provides the stack and frame as defined and initializes the frame and stack pointer and also the XRAM access, if present.

If structured constants are placed into the BootBlock they also become initialised. Please note that with *BootVectors* import this function is always executed and **must not be called**.

*Procedure* `BootRestart;`

Also the application can execute a boot operation by calling this procedure. But this entry is only available if *BootVectors* are imported !

## 4.20.3 Constants

In conjunction with complex Boot drivers sometimes it is necessary to have “nonvolatile“ constants and structured constants in the BootBlock which can be accessed at any time by the Boot driver.

These constants then must be defined in the BootBlock. They don't reside in the common Flash constant pool but in the BootBlock. If *BootVectors* are imported then the structured constants will be automatically copied to their correct place in the RAM. Without the *BootVectors* import the function *Boot\_Init* must provide this.

All constants are also accessible by the application.  
Absolute Constants in the BootBlock are allowed.

### Important

Working with these constants in the BootBlock can be dangerous. Because the Flash becomes erased or rewritten at this time in most cases system functions are invalid or have changed their absolute address. So many operations with constants may be impossible in the BootBlock. An exact analysis of the involved system calls (CALL or RCALL) then is necessary. The same is true with the handling of variables



# AVRco Compiler-Manual

## **type**

```
tTestrec = record
    b1 : byte;
    b2 : byte;
end;
```

```
{$PHASE BootBlock $0F000}
```

## **structconst**

```
abcde : array[0..3] of byte = ($12, $34, $56, $78);
TestrecS : tTestrec = (b1 : $12; b2 : $34);
vStr : string = 'abcdef';
```

## **const**

```
xyz : array[0..3] of byte = ($12, $34, $56, $78);
cStr : string = '12345';
TestrecC : tTestrec = (b1 : $12; b2 : $34);
```

```
Procedure BootTest;
```

## **begin**

```
if abcde[0] <> xyz[0] then ...
endif;
if TestrecS.b1 <> TestRecC.b2 then ...
endif;
if cStr[1] <> 'a' then ...
endif;
if vStr[1] <> cStr[1] then ...
endif;
```

## 4.20.4 Example Program

The following shows a small example how to use boot vector switching and the interrupt routines. Then the resulting vector tables are listed.

```
program AVR_BootVectors;
```

```
{$BootRst $0F000} {reset jumps to here}
{$NOSHADOW}
```

```
Device = mega128, VCC=5;
```

```
Import SysTick;
```

```
From System Import BootVectors;
```

## **Define**

```
ProcClock = 8000000; {Hertz}
SysTick = 10; {msec}
StackSize = $0020, iData;
FrameSize = $0040, iData;
```

## Implementation

```

{$IDATA}
{-----}
{ Var Declarations }
var
  iii : byte;

{-----}
{ functions }

{$PHASE BootBlock $0F000}           // start of boot area
Procedure BootTest;                // this is the first code building
begin                               // part of the boot block
  SetVectTabBoot(true);             // this is the absolute reset entry
  // vector table in Boot area
  iii:= 0;
  EnableInts;
  repeat                             // wait for pin interrupts
  until iii > 3;                       // which increments iii
  DisableInts;                       // start over to the MAIN
  // enable vector relocation
  SetVectTabBoot(false);            // vector table on address 0
  ASM: JMP SYSTEM.VectTab;          // absolute address $0000 = RESET vector
end;

Interrupt INT0_BOOT;                // only for the Boot vector table
begin                               // forced by the extension "_boot"
  inc(iii);
end;

Interrupt INT1;                     // for both vector tables
begin                               // because there is no extension
  inc(iii);                           // but it is still located in the boot area
end;
{$DEPHASE BootBlock}                // end of boot area

Interrupt INT0;                     // only for the basic vector table
begin                               // because it is outside of the boot block
  inc(iii);
end;

{-----}
{ Main Program }
begin
  EnableInts;
  loop
  nop;
  endloop;
end AVR_BootVectors.

```



# AVRco Compiler-Manual

## BOOT Area

```
0243 F000 .PHASE 01E000h BOOT
0244 F046 .ORG 01E08Ch, BOOT
0245 F046 SYSTEM.$BOOT_ENTRY:
0246 F046
0247 F046 .FUNC BootTest, 42, 00020h
0248 F046 AVR_BootVectors.BootTest:
0249 F046 .BLOCK 45
0250 F046 .LINE 45
0251 F046 EF1F LDI _ACCA, 0FFh
0252 F047 2311 TST _ACCA
0253 F048 F009 BREQ AVR_BootVectors.L0000
0254 F049 E012 LDI _ACCA, 2
0255 F04A AVR_BootVectors.L0000:
0256 F04A 91000055 LDS _ACCB, MCUCR
0257 F04C 7F0D CBR _ACCB, 02h
0258 F04D 6001 SBR _ACCB, 01h
0259 F04E 93000055 STS MCUCR, _ACCB
0260 F050 7F0E CBR _ACCB, 01h
0261 F051 2B01 OR _ACCB, _ACCA
0262 F052 93000055 STS MCUCR, _ACCB
0263 F054 .LINE 47
0264 F054 E010 LDI _ACCA, 000h
0265 F055 93100101 STS AVR_BOOTVECTORS.III, _ACCA
0266 F057 .LINE 48
0267 F057 E810 LDI _ACCA, 1 SHLB IntFlag
0268 F058 2A21 OR Flags, _ACCA
0269 F059 9478 SEI
0270 F05A AVR_BootVectors.L0001:
0271 F05A .BLOCK 50
0272 F05A .ENDBLOCK 50
0273 F05A .LINE 50
0274 F05A AVR_BootVectors.L0002:
0275 F05A 91100101 LDS _ACCA, AVR_BootVectors.iii
0276 F05C 3013 CPI _ACCA, 003h
0277 F05D E010 LDI _ACCA, 0h
0278 F05E F011 BREQ AVR_BootVectors.L0004
0279 F05F F008 BRLO AVR_BootVectors.L0004
0280 F060 EF1F SER _ACCA
0281 F061 AVR_BootVectors.L0004:
0282 F061 2311 TST _ACCA
0283 F065 .BRANCH 4, AVR_BootVectors.L0005
0284 F062 F411 BRNE AVR_BootVectors.L0005
0285 F05A .BRANCH 20, AVR_BootVectors.L0001
0286 F063 940CF05A JMP AVR_BootVectors.L0001
0287 F065 AVR_BootVectors.L0005:
0288 F065 AVR_BootVectors.L0003:
0289 F065 .LINE 52
0290 F065 94F8 CLI
0291 F066 E71F LDI _ACCA, 0FEH ROLB IntFlag
0292 F067 2221 AND Flags, _ACCA
0293 F068 .LINE 54
0294 F068 E010 LDI _ACCA, 000h
0295 F069 2311 TST _ACCA
0296 F06A F009 BREQ AVR_BootVectors.L0006
0297 F06B E012 LDI _ACCA, 2
0298 F06C AVR_BootVectors.L0006:
0299 F06C 91000055 LDS _ACCB, MCUCR
0300 F06E 7F0D CBR _ACCB, 02h
0301 F06F 6001 SBR _ACCB, 01h
0302 F070 93000055 STS MCUCR, _ACCB
0303 F072 7F0E CBR _ACCB, 01h
0304 F073 2B01 OR _ACCB, _ACCA
0305 F074 93000055 STS MCUCR, _ACCB
0306 F076 .LINE 55
0307 F076 940C0000 JMP SYSTEM.VectTab; // absolute address $0000
// = RESET vector
0308 F078 .ENDBLOCK 56
0309 F078 AVR_BootVectors.BootTest_X:
0310 F078 .LINE 56
0311 0000 .BRANCH 19
0312 F078 9508 RET
0313 F079 .ENDFUNC 56
0314 F079
```



```

0315 F079                .FUNC      INTERRUPT_INT0_BOOT, 58, 00020h
0316 F079                AVR_BootVectors.INTERRUPT_INT0_BOOT:
0317 F079                94E8                CLT
0318 F07A                F827                BLD      Flags, IntFlag
0319 F07B                .BLOCK      60
0320 F07B                .LINE      60
0321 F07B                91100101          LDS      _ACCA, AVR_BootVectors.iii
0322 F07D                9513                INC      _ACCA
0323 F07E                93100101          STS      AVR_BootVectors.iii, _ACCA
0324 F080                .ENDBLOCK 61
0325 F080                AVR_BootVectors.INTERRUPT_INT0_BOOT_X:
0326 F080                .LINE      61
0327 F080                9468                SET
0328 F081                F827                BLD      Flags, IntFlag
0329 0000                .BRANCH    19
0330 F082                9508                RET
0331 F083                .ENDFUNC   61
0332 F083
0333 F083                .FUNC      INTERRUPT_INT1, 63, 00020h
0334 F083                AVR_BootVectors.INTERRUPT_INT1:
0335 F083                94E8                CLT
0336 F084                F827                BLD      Flags, IntFlag
0337 F085                .BLOCK      65
0338 F085                .LINE      65
0339 F085                91100101          LDS      _ACCA, AVR_BootVectors.iii
0340 F087                9513                INC      _ACCA
0341 F088                93100101          STS      AVR_BootVectors.iii, _ACCA
0342 F08A                .ENDBLOCK 66
0343 F08A                AVR_BootVectors.INTERRUPT_INT1_X:
0344 F08A                .LINE      66
0345 F08A                9468                SET
0346 F08B                F827                BLD      Flags, IntFlag
0347 0000                .BRANCH    19
0348 F08C                9508                RET
0349 F08D                .ENDFUNC   66
0350 F08D
0351 F08D                ; ===== String-constant tables =====
0352 F08D
0353 F08D                SYSTEM.BootIntErr:
0354 F08D                9518                RETI
0355 F08E
0356 F08E                SYSTEM._Boot_Init:
0357 F08E                E011                LDI      _ACCA, 01h
0358 F08F                BF1B                OUT      RAMPZ, _ACCA
0359 F090
0360 F090                E110                LDI      _ACCA, 010FFh SHRB 8
0361 F091                EF0F                LDI      _ACCB, 010FFh AND 0FFh
0362 F092                BF1E                OUT      sph, _ACCA
0363 F093                BF0D                OUT      spl, _ACCB
0364 F094                E1D0                LDI      _FPTRHI, 010BFh SHRB 8
0365 F095                EBCF                LDI      _FRAMEPTR, 010BFh AND 0FFh
0366 F096
0367 F096                ; no Peripheral sram-waits
0368 F096                B715                IN       _ACCA, mcucr
0369 F097                7B1F                CBR     _ACCA, 040h
0370 F098                BF15                OUT     mcucr, _ACCA
0371 F046
0372 F046                .BRANCH    20,AVR_BootVectors.BootTest
0373 F099                CFAC                RJMP    AVR_BootVectors.BootTest
0374 F09A
0375 F09A                SYSTEM.$INTERRUPT_INT0_BOOT:
0376 F09A                93EF                PUSH    _ACCCLO
0377 F09B                93FF                PUSH    _ACCCHI
0378 F09C                930F                PUSH    _ACCB
0379 F09D                931F                PUSH    _ACCA
0380 F09E                B71F                IN      _ACCA, SREG
0381 F09F                931F                PUSH    _ACCA
0382 F079                .BRANCH    17,AVR_BootVectors.INTERRUPT_INT0_BOOT
0383 F0A0                940EF079          CALL    AVR_BootVectors.INTERRUPT_INT0_BOOT
0384 F0A2                911F                POP     _ACCA
0385 F0A3                BF1F                OUT     SREG, _ACCA
0386 F0A4                911F                POP     _ACCA
0387 F0A5                910F                POP     _ACCB
0388 F0A6                91FF                POP     _ACCCHI
0389 F0A7                91EF                POP     _ACCCLO
0390 F0A8                9518                RETI
0391 F0A9

```



# AVRco Compiler-Manual

```
0392 F0A9          SYSTEM.$INTERRUPT_INT1:
0393 F0A9          93EF          PUSH     _ACCCLO
0394 F0AA          93FF          PUSH     _ACCCHI
0395 F0AB          930F          PUSH     _ACCB
0396 F0AC          931F          PUSH     _ACCA
0397 F0AD          B71F          IN      _ACCA, SREG
0398 F0AE          931F          PUSH     _ACCA
0399 F083          .BRANCH    17,AVR_BootVectors.INTERRUPT_INT1
0400 F0AF          940EF083     CALL    AVR_BootVectors.INTERRUPT_INT1
0401 F0B1          911F          POP     _ACCA
0402 F0B2          BF1F          OUT    SREG, _ACCA
0403 F0B3          911F          POP     _ACCA
0404 F0B4          910F          POP     _ACCB
0405 F0B5          91FF          POP     _ACCCHI
0406 F0B6          91EF          POP     _ACCCLO
0407 F0B7          9518          RETI
0408 F000
0409 F000
0410 F000          .ORG      01E000h, VECTTABB
0411 F000          .VECTTAB_B
0412 F000          SYSTEM.VectTab_B:
0413 F002          940CF08E     JMP     SYSTEM._Boot_Init
0414 F004          940CF0A9     JMP     SYSTEM.$INTERRUPT_INT0_BOOT
0415 F006          940CF08D     JMP     SYSTEM.$INTERRUPT_INT1
0416 F008          940CF08D     JMP     SYSTEM.BootIntErr
0417 F00A          940CF08D     JMP     SYSTEM.BootIntErr
0418 F00C          940CF08D     JMP     SYSTEM.BootIntErr
0419 F00E          940CF08D     JMP     SYSTEM.BootIntErr
0420 F010          940CF08D     JMP     SYSTEM.BootIntErr
0421 F012          940CF08D     JMP     SYSTEM.BootIntErr
0422 F014          940CF08D     JMP     SYSTEM.BootIntErr
0423 F016          940CF08D     JMP     SYSTEM.BootIntErr
0424 F018          940CF08D     JMP     SYSTEM.BootIntErr
0425 F01A          940CF08D     JMP     SYSTEM.BootIntErr
0426 F01C          940CF08D     JMP     SYSTEM.BootIntErr
0427 F01E          940CF08D     JMP     SYSTEM.BootIntErr
0428 F020          940CF08D     JMP     SYSTEM.BootIntErr
0429 F022          940CF08D     JMP     SYSTEM.BootIntErr
0430 F024          940CF08D     JMP     SYSTEM.BootIntErr
0431 F026          940CF08D     JMP     SYSTEM.BootIntErr
0432 F028          940CF08D     JMP     SYSTEM.BootIntErr
0433 F02A          940CF08D     JMP     SYSTEM.BootIntErr
0434 F02C          940CF08D     JMP     SYSTEM.BootIntErr
0435 F02E          940CF08D     JMP     SYSTEM.BootIntErr
0436 F030          940CF08D     JMP     SYSTEM.BootIntErr
0437 F032          940CF08D     JMP     SYSTEM.BootIntErr
0438 F034          940CF08D     JMP     SYSTEM.BootIntErr
0439 F036          940CF08D     JMP     SYSTEM.BootIntErr
0440 F038          940CF08D     JMP     SYSTEM.BootIntErr
0441 F03A          940CF08D     JMP     SYSTEM.BootIntErr
0442 F03C          940CF08D     JMP     SYSTEM.BootIntErr
0443 F03E          940CF08D     JMP     SYSTEM.BootIntErr
0444 F040          940CF08D     JMP     SYSTEM.BootIntErr
0445 F042          940CF08D     JMP     SYSTEM.BootIntErr
0446 F044          940CF08D     JMP     SYSTEM.BootIntErr
0447 F046
0448 F046          .VECTTAB_B
0449 0046          .DEPHASE
```

## Application Area

```

0749 8004                .ENDCODE
0750 0000                .ORG          0, VECTTAB
0751 0000                .VECTTAB
0752 0000                SYSTEM.VectTab:
0753 0000 940C0065        JMP          SYSTEM.RESET
0754 0002 940C00A9        JMP          SYSTEM.$INTERRUPT_INT0
0755 0004 940CF0A9        JMP          SYSTEM.$INTERRUPT_INT1
0756 0006 940C00FB        JMP          SYSTEM.DefIntErr
0757 0008 940C00FB        JMP          SYSTEM.DefIntErr
0758 000A 940C00FB        JMP          SYSTEM.DefIntErr
0759 000C 940C00FB        JMP          SYSTEM.DefIntErr
0760 000E 940C00FB        JMP          SYSTEM.DefIntErr
0761 0010 940C00FB        JMP          SYSTEM.DefIntErr
0762 0012 940C00FB        JMP          SYSTEM.DefIntErr
0763 0014 940C00FB        JMP          SYSTEM.DefIntErr
0764 0016 940C00FB        JMP          SYSTEM.DefIntErr
0765 0018 940C00FB        JMP          SYSTEM.DefIntErr
0766 001A 940C00FB        JMP          SYSTEM.DefIntErr
0767 001C 940C00FB        JMP          SYSTEM.DefIntErr
0768 001E 940C00FB        JMP          SYSTEM.DefIntErr
0769 0020 940C0093        JMP          SYSTEM.$INTERRUPT_TIMER0
0770 0022 940C00FB        JMP          SYSTEM.DefIntErr
0771 0024 940C00FB        JMP          SYSTEM.DefIntErr
0772 0026 940C00FB        JMP          SYSTEM.DefIntErr
0773 0028 940C00FB        JMP          SYSTEM.DefIntErr
0774 002A 940C00FB        JMP          SYSTEM.DefIntErr
0775 002C 940C00FB        JMP          SYSTEM.DefIntErr
0776 002E 940C00FB        JMP          SYSTEM.DefIntErr
0777 0030 940C00FB        JMP          SYSTEM.DefIntErr
0778 0032 940C00FB        JMP          SYSTEM.DefIntErr
0779 0034 940C00FB        JMP          SYSTEM.DefIntErr
0780 0036 940C00FB        JMP          SYSTEM.DefIntErr
0781 0038 940C00FB        JMP          SYSTEM.DefIntErr
0782 003A 940C00FB        JMP          SYSTEM.DefIntErr
0783 003C 940C00FB        JMP          SYSTEM.DefIntErr
0784 003E 940C00FB        JMP          SYSTEM.DefIntErr
0785 0040 940C00FB        JMP          SYSTEM.DefIntErr
0786 0042 940C00FB        JMP          SYSTEM.DefIntErr
0787 0044 940C00FB        JMP          SYSTEM.DefIntErr
0788 0046                .VECTTABE
0789 0046                SYSTEM.ENDPROG:
0790 0046

```

## Sample Program

An example can be found in the directory `..E-Lab\AVRco\Demos\BootVectors`



## 4.21 BOOT TRAPS

If essential parts of the system are placed into the Boot section and these functions must also be accessible from the application section then there is the problem that with a download of a new application part through the Boot Loader the new application can use new but illegal physical addresses in the boot section.

A simple call of functions or procedures out of the application can point to wrong or illegal addresses in the code or the boot block with catastrophic results. In order to avoid this there must be a special interface which was introduced many years ago by CP/M and DOS for entering the BIOS.

Here a function is never called directly but indirectly through a jump table which is placed at the beginning of this area on a well known and fixed address. By this theoretically both parts can be changed without leading to dramatic results. However our table is unchangeable at runtime as is the whole boot area.

### How does it work?

The system builds a jump table with 16 entries after the Boot Vector Table. If no Boot Vectors are used this table is directly placed at the Boot entry address. Each function or procedure in the Boot area which has the "**TRAP**" attribute is entered into this table by its address.

A call of such a function or procedure from the application section then is not built as usual with a "CALL Procedure" but the address of this function is read out of this table and used for an ICALL operation. However a call of such a Trap function from within the Boot area is handled as usual with a standard CALL.

### 4.21.1 Implementation of the Boot Traps

#### Imports

The Boot Trap Handler must be imported.

```
From System Import Traps {,BootVectors};
```

#### Defines

There is no special Boot Trap Handler define.

```
{$PHASE BootBlock $0F00}
```

```
...
```

```
Procedure TestTrapP(b : byte; w : word); Trap;
```

```
begin
```

```
  ww:= w;
```

```
end;
```

```
Procedure TestTrap; Trap;
```

```
begin
```

```
  TestTrapP (12, 1234);           // procedure is called as usual
```

```
end;
```

```
{$DEPHASE BootBlock}
```

```
// MAIN
```

```
begin
```

```
  TestTrap;                       // functions are called through the jump table
```

```
  TestTrap;
```

```
  bo:= TestTrapF (12, 1234);
```

```
  TestTrapP (12, 1234);
```

```
...
```

## 4.22 Inheritance

The basis for objects is what is known as inheritance. A new object takes on all the properties of an existing object and extends the new one with additional properties. Objects are special records. If it is possible to import an existing record into a new one simple objects can be constructed.

The import respectively the inheritance of an existing record into a new one must be done with the use of

*inherit RecordName;*

**type**

```
tFirstRec = record
    bb : byte;
    ww : word;
    pp : pointer;
end;

tObjRec = record
    inherit firstRec; // import of firstRec
    ii : integer;
    pro : procedure;
end;
```

"ObjRec" inherits the properties of "firstRec". Please note that the inherit instruction must always be the **first** declaration of the new record.

Internally the new records is constructed in this way:

```
tObjRec = record
    bb : byte; // inherited
    ww : word; // inherited
    pp : pointer; // inherited
    ii : integer;
    pro : procedure;
end;
```

It must be clear that no identifier can be used twice, all must be unique. A simple application of such a construct is the hiding of the inherited part:

**type**

```
tpFirstRec = pointer to tFirstRec;
tpObjRec = pointer to tObjRec;
```

**var**

```
ObjRec : tObjRec;
pFirstRec : tpFirstRec;
pObjRec : tpObjRec;
...
pFirstRec:= tpFirstRec (@ObjRec); // only FirstRec is visible
pObjRec:= @ObjRec; // all is visible
```

## 5 Multi-Tasking Programming

### 5.1 Introduction

With an Embedded Application (Single-Chip application) often there is the problem, that several jobs should be done at the same time. For example the characters of a serial interface should be fetched, checked and perhaps they should be converted from hex into an integer. At the same time ports should be watched by limit-switches or a LED should flash. Additionally a measurement value should be gathered by a pot and this value should be passed as a control output to an external controller. And the controller should calculate an output value in a fixed time grid.

So the programmer has the problem with all these targets to do all things concurrently. The programmer is in the difficult situation to watch several processes at the same time, whereby he must take care that all functions run **concurrently and independently**.

With simple time-loops etc. this problem cannot be solved except maybe with tricks, which make the program inflexible and bovine.

So a solution is needed, which makes it possible to distribute the jobs, that they frequently get a chance to work, but do not block other jobs. Such a system is called **Multi-Tasking**, whereby a task is a job/assignment.

The terms **tasks** or **processes** are often treated as synonyms in literature, but within the AVRco they are distinguished. Details see below.

In connection with Multi-Tasking there is always associated the term **Real-Time**. Strictly speaking real-time has nothing to do with multi-tasking. Real-time means to responding to external events as quickly as possible. In general this is achieved via interrupts. But the reaction-time is not predefined anywhere. Depending on the requirement this can mean the system has to react within microseconds or sometimes milliseconds. The application basically determines what is real-time and what is too slow.

### 5.2 Principle of Operation

Multi-Tasking or Multi-Processing is defined as the ability to process several jobs/tasks quasi-parallel, it seems as if they run at the same time. Because it is not possible for a processor to read and process several machine commands at the same time, this must be done sequentially, i.e. one after another.

This sequential processing of jobs (calling of jobs, handout of run-time, switch over to the next job) is done by what is known as the **scheduler**. The scheduler is processed in a timer-interrupt (SysTick). It checks if the running process/task has consumed its temporary run-time. If it is consumed, the working-register, stackpointer, flags etc. are saved in a memory area, which is assigned to this process, and the next process or task is called.

This method is also called **Round-Robin**, because the processes are done essentially in a circle. Other possible methods are not considered, because the administrative outlay is too big.

The scheduler observes the state of the separate tasks. This is the *priority*, which defines the time-slice, and waiting flags (pipe, semaphore, sleep), which switch off a process temporarily. *Suspend* switches a process completely off and *lock* ties the CPU completely to this process.

## 5.2.1 Processes and Tasks

Processes and tasks are essentially independent programs within an application, which are able to run independently from other program parts (e.g. main). Processes can not be called like functions or procedures. Instead of this they are called periodically by the scheduler. If processes have been imported, the main program runs as a process, too, and also has a priority, name(Main\_Proc) and process-ID(0).

If there are several processes/tasks within a program, the processes are done quasi-parallel, i.e. it seems as if all processes are done at the same time = **Multi-Tasking**. So an apparently **parallel processing** for example of events or data is achieved, although they are always processed sequentially, i.e. one after another.

A process practically runs for ever, only interrupted by interrupts and other processes and tasks. The 'begin' and 'end' limits a process and its statements. Because processes cannot be called like functions, they do not have any passing parameters or results.

With the first call of a process by the scheduler, it is started with the statement which follows directly after the 'begin' statement. Then all statements are processed until 'end' is reached, perhaps interrupted by a **task change** by the scheduler (Switch over to another process/task). If the 'end' is reached, it is automatically continued with the first statement after 'begin'. So a process runs continuously in a 'circle' without ending. The programmer does not need program a loop, because the jump back to the beginning (begin) happens automatically.

This is the essential difference to a task. With **every** call of a task by the scheduler it is started with the statement which follows immediately after the 'begin' statement. Then all statements are processed until 'end' statement. If the 'end' is not reached within a system tick, the task is **interrupted** by a **task change** by the scheduler (Switch over to another process/task). So the task never reaches the 'end', if it's required run-time from 'begin' to 'end' is longer than a system tick. The run-time may **never** be longer than a system tick. Similar conditions are also valid for interrupts. A timer-interrupt-service-routine for example, should never take more time than the period between two interrupts.

If the 'end' is reached, the control is automatically passed to the scheduler, which now activates the next process or task. In contrast to a process a task runs from 'begin' until 'end' with every call through the scheduler and then aborts.

## 5.2.2 Priority

The behavior of a process/task is controlled by a number of corresp. functions and procedures. An essential parameter is priority.

With **priority** a part of the globally available run-time is placed at disposal to a process. The higher the value of priority is, the more run-time is at disposal. At the same time priority predefines the number of system ticks, which are completely available for the process in a piece. The proportional run-time of the total time in % is calculated by: Priority / Sum of all priorities.

Assumed there is only the process 'DoTheJob' and it has the priority 10 and Main Priority is 5 then is valid: run-time =  $10 / (5 + 10) = 66\%$ . But the exact run-time can only be predefined if no process is suspended or locked and if no ProcessWaits etc. exist. In practice the proportional run-time can only be estimated.

In contrast to a process the calling interval of the **task** is predefined by **priority**. The lower the value of priority is, the more often is the calling of the task. Assumed the task 'RunPid' has the priority 10, so it is called every 10. Systick. So it is established, that the period between two callings is always 10 ticks.

## **Note:**

If there are several **tasks**, and if it is possible, that several of them are active, pay absolutely attention that all priorities have a **common denominator**.

I.e. all priorities must be a multiple of 2, for example. If this condition is not met, so there are irregular calling intervals, i.e. the period between two callings is not constant any longer. Further the lowest priority should be higher than (count of all tasks + count of all processes) but at least higher than the task count.

A process/task is able to take over the CPU completely by **lock**, so apart from itself only interrupts are running. This state is terminated by **unlock**.

If a process/task finds out, that it has nothing to do at the moment, there should be no waste of run-time by waitloops or delays. There are several possibilities to pass the control to other processes:

With **Schedule** the process/task is interrupted immediately, but is enqueued again into the waiting loop. With **Sleep** a process/task is able to switch itself off for a certain number of system ticks.

With **Suspend** a process/task switches off. It is not able to switch itself active again. This must be done by another process/task or by the main program with **resume**.

Because the communication between tasks/processes is made by pipes and semaphores, the task may suspend itself by calling **WaitSema** or **WaitPipe**. The process/task then becomes active if there is data in the specified semaphore or pipe. It is also possible that **RxBuffer** (RxBuffer1, -2, -3) is specified as a pipe.

The process/task is interrupted directly after an above mentioned instruction.

### **5.2.2.1 Default Priorities**

without explicit priority setting the priorities are

Main:	Priority 5
Prozess:	Priority 3
Task:	Priority 5

## **5.3 Optimal Multi-Tasking**

With a multitasking application accurate planning is necessary.

1. With the right strategy and a good partitioning of the jobs into separate tasks and processes fast program processing is achieved.
2. In spite of the overhead (ca. 330 cycles), which is caused by the scheduler, a multitask program runs much faster than a normal programmed program, if there is suitable use of schedule, sleep, WaitSema, WaitPipe and suspend.
3. The right distribution of the priorities decides the reaction-time of the separate processes and hence the real-time capability. If the priorities are dynamic, i.e. they are changed for the needs of the application during the program run, an additional improvement of the system is achieved.



## 6 Optimization

### 6.1 Library

The compiler optimizes the library calls completely, i.e. only those library functions, which are used, are imported and occupy code space and possible RAM space (variable). So there is no waste of resources.

#### 6.1.1 Variable

The Pascal compiler has no variable optimization at the moment. That means that a variable, which has been declared, also physically occupies space. If this variable is not used within the program, so the space is wasted. It is planned about to introduce a variable optimization later, which eliminates the unused variable.

The programmer must take care to delete the unused variables and to treat the resources with care. The compiler switch `{W+}` helps.

#### 6.1.2 Constant

A constant declaration like “**Const**  $x = 123;$ “ occupies no resources within the CPU, whether this constant is used or not.

Constant expressions for example “ $a := (5 * 10) + (24 \text{ div } 2);$ “ are parsed by the compiler, if possible. This is called constant folding. The result of the above expression is “62“. The compiler then uses the constant result and passes it to the assembler like: `LDI _ACCA, 62`

#### 6.1.3 Runtime

The generated program within many compilers is what is known as stack machine. All passing parameters and provisional results within terms are stored in stacks and frames. Often there are unnecessary PUSH and POP operations connected with it. Also variables are often loaded into the working register, although this variable is already in this register. This is caused by the formalism, which exists in every compiler.

A run-time optimization realizes those unnecessary operations and eliminates them. This is built into this compiler. There are many further possibilities for optimization, which will be reserved for future releases.

## 6.2 Highly Optimizing?

Many compilers attire themselves (sometimes even with justification) with the attribute “highly optimizing“. A very hard optimization often has two sides. If, for example, all actual variables are within the register set and are processed there, there is no possibility for an interrupt-routine or a multitask to get an access to these actual values! Further this routine cannot be recursive, but at least it must try to save the registers. Highly optimized code is generally not reentrant also (abort of a routine by an interrupt or multitasking and a calling of it within the interrupt task).

Because of that it is necessary within certain compilers that certain procedures or functions require a compiler switch to switch off the optimization.

Beginners as well as experienced users often forget it, and so sometimes there is a crash that cannot be reproduced. Such a system is not easy to clean. For beginners it is only possible if essential optimizations are switched off.

The AVRco generally works with stack frames, it is slower, but extremely secure. All system- and user-functions/procedures are generally reentrant as well as recursive. The recursion depth is only limited by the Ram and the stack size.



# AVRco Compiler-Manual

Within the AVRco the floating point library is reentrant, i.e. it could be used within interrupts and recursions, this is not the case with all compilers. (With interrupts save all registers!).

Although the speed and a compact code are necessary targets of a compiler, within the AVRco the most necessary thing is the reliability requirement and the optimization was secondarily, but this does not mean that optimization is no theme. As mentioned above in future there will be better optimizing versions, too.

## **The most compact and fastest code**

is reached within every system by the system library. The more functions are within the system, the less has the programmer to formulate in a high-level language. System functions are completely written in assembler, so they are highly efficient, i.e. fast and short. The "Librarian", which is contained in some compilers, generates out of the high-level language a linkable library. The generated code is exactly the same as the code, which is made out of the statements by the compiler, if they are directly included in the program. So there is no profit that way.

An I<sup>2</sup>C-Bus has to be formulated in a high-level language by the programmer within most of the systems. This code is many times bigger and slower than the corresponding system functions. By the import of several system functions, and there are many of them in the AVRco, the moderate optimization of the AVRco is made much better.

The best optimizer is the programmer himself. There can be an optimal code, if multitasking is used and local variables are used sparingly.

The generated code of the AVRco (also in connection with Multi-Tasking) can also feel free with the compare to more complex applications.

A "foolproof" program is the target of the AVRco. A highest optimized program can only be reached with a careful using of the compiler switches, and so there can be many mistakes/errors!

As a matter of fact a little bit bigger processor with larger ram/rom gives a better improvement than an optimization, and often it costs just a little bit more than a weaker version, even the increase of efficient periphery (ports, timer, interfaces) is better.

## **6.3 The "Merlin Optimizer"**

The Merlin Optimizer is a Plug-In for the Standard and Profi Version of the AVRco System. It reduces the code size between 5..20%.

The Optimizer is called from inside the IDE with HotKey or SpeedButton.

The complete sequence "compile" – "optimize" – "assemble" needs also only a single mouse click.

From Version 4 of the AVRco (Standard and Profi) the Merlin Optimizer is included. For Version 3 the Optimizer has to be purchased separately.

## 7 Compiler Switches

Compiler Switches are serving to control the behavior of the compiler. These switches are components of the source.

A switch is started with a { and a following \$ without a space. Immediately after the \$ the switch name has to follow. Instruction for further parameters has to be done like the usual conventions.

The switches must start in the first column of a line. The same line may not contain any statements.

Syntax: **{*SWITCH* [arg] }**

Each Import and each Device Define is added to the compiler switch list. This means that:

*Import LCDport, ...*

is treated like: {\$DEFINE LCDPORT} and

*Define ProcClock = 8000000;*

is treated like: {\$DEFINE PROCCLOCK}

### 7.1 Memory Administration

{\$DATA} {\$SIDATA} {\$SIDATA1} {\$PDATA} {\$XDATA} {\$XDATA1}..{\$XDATA4} {\$EEPROM} {\$SUDATA}

Selects the memory page of the CPU. Most of the small processors are only able to reach a small memory range if there is a direct address. Possible additional existing memory needs extra processing. This is specially valid for the external memory, if it existing. The switch assigns the appropriate area to the following variables.

**{\$DATA}**

\$DATA assigns the area, which is found in the processor control file (xxx.dsc) within DATA, to all following variable declarations (also **structconst**). With the AVR this area is from \$04 upto \$1F. The defined variables now are progressed placed from \$04. If another switch of this type occurs later, the following variables are processed analogously. Variables in the area \$DATA are always reachable with very short and fast machine commands.

**{\$PDATA}**

\$PDATA assigns the area, which is found in the processor control file (xxx.dsc) within PDATA, to all following variable declarations. With the AVR 8515 this area is from \$20 upto \$5F. The defined variables now are progressed placed from \$20. PDATA is reserved for an IO-area, if existing.

\$PDATA mostly is accessed with special machine commands. Within the definition of variables in this area, the compiler should not allocate the address, but the programmer must/should specify the required address of every variable:

*Var Port1[\$35] : byte;*



# AVRco Compiler-Manual

## **{\$IDATA}**

**\$IDATA** assigns the area, which is found in the processor control file (xxx.dsc) within IDATA, to all following variable declarations (also **structconst**). With the AVR 8515 this area is from \$60 upto \$25F. The defined variables now are progressively placed from \$60. If another switch of this type is found, with the following variables are proceeded analogously. Variables in the area **\$IDATA** are always accessible with longer and slower machine commands.

The iData normally includes the whole internal SRAM of the CPU. If necessary this area can be divided into two parts. This is supported with a Define. With **iDATA1** the internal RAM is divided into two parts. The necessary parameter then defines the end of the first (lower) part and on the other hand the start address of the second part. (\* REV4 \*)

```
Define iData1 = $800;
```

Variables now can be placed into the iData or ldata1 area.

Example:

```
{$IDATA1}  
Var  
Test1 : word;
```

```
{$IDATA}  
Var  
Test : word;
```

## **{\$XDATA} {\$XDATA1} {\$XDATA2} {\$XDATA3} {\$XDATA4}**

**\$XDATA** assigns the area, which is **Defined** with XDATA, to all following variable declarations (also **structconst**). XDATA is external memory, which is only available within the bigger types. If another switch of this type is found, with the following variables are proceeded analogously. Variables in the area **\$XDATA** are always accessible with longer and slower machine commands. Often the CPU uses additional Wait states.

```
Define xDataWaits = nn; // 0..3
```

Select the xData Waits for e.g. mega8515, mega162, mega64 and mega128.

```
Define XDATA1 = $8000, $80FF, NoInit;
```

Each XDATA can be assigned to **NoInit** within the XDATA definition.

## **{\$EEPROM}**

**\$EEPROM** assigns the area, which is found in the processor control file (xxx.dsc) within EEPROM, to all following variable declarations (also **structconst**). EEPROM only can be a chip intern memory.

The EEPROM normally includes the whole internal EEPROM of the CPU. If necessary this area can be divided into two parts. This is supported with a Define. With **EEPROM1** the internal EEPROM is divided into two parts. The necessary parameter then defines the end of the first (lower) part and on the other hand the start address of the second part. (\* REV4 \*)

```
Define EEPROM1 = $800;
```

Variables now can be placed into the EEPROM or EEPROM1 area.

Example:

```
{$EEPROM1}  
Var  
Test1 : word;
```

```
{$EEPROM}  
Var  
Test : word;
```

## **{ \$UDATA }**

\$UDATA assigns the area, which is **Defined** in the definitions section with UserData (UserDevice), to all following variable declarations. This data area resides in an external device which is not accessible through the normal CPU-addressings, e.g. a serial EEPROM. The programmer must supply a device driver. See also paragraph **Device Driver** in the *Standard Driver Manual*.

The variables always build up from the lower to the higher address. If the memory areas are changed by a new switch, so it is continued with the actual memory at **last specified address**.

## **Memory Initialization**

The memory areas \$DATA, \$IDATA and the \$XDATA section basically are initialized to \$00. This initialization can be disabled for the \$XDATA areas at the definition time by the attribute **NOINIT**. This is valid for the entire area. The following compiler switch can be used for the \$DATA and \$IDATA sections:

## **{ \$NOINIT }**

Switch for the \$DATA and \$IDATA area. The following variables of this section upto it's end are not initialized, i.e. they are not filled with zeros.

But you should note that the standard initialization is done in a very fast loop over a single block.

If you insert Noinit variables in between this process may take much more time.

You should group these var definitions to avoid extensive fragmentation.

## **Variables outside any area**

If variable has to be located outside any defined area, the compiler generates an error. The error can be suppressed with the following Compiler switch:

## **{ \$NORAMCHECK }**

The following variable is not checked for a valid RAM area.

```
{ $IDATA }
```

```
{ $NORAMCHECK }
```

```
var
```

```
Extreme[ @$FFFF ] : byte;
```

## **{ \$PHASE } { \$DEPHASE }**

Switches to a fixed **WORD**-Address in Flash and back to the standard code page.

```
{ $PHASE $1E00 }; places the following code at addr $1E00 and up.
```

```
{ $DEPHASE }; switches the address generation back to the default page.
```



# AVRco Compiler-Manual

## 7.1.1 Considerations about Memory Usage

The main difference between the separate memory areas is obviously the required time and the program space consumption, caused by the according addressing types of the CPU. Within time critical applications the programmer should assign his variables corresponding to the number of accesses to the variables.

Default: {\$DATA}

**var**

```
b1 : byte;      {$06}
ch1 : char;     {$07}
w1 : word;     {$08}
bool1 : boolean; {$0A}
```

{\$IDATA}

**var**

```
b2 : byte;      {$60}
ch2 : char;     {$61}
w2 : word;     {$62}
bool2 : boolean; {$64}
```

{\$PDATA}

**var**

```
Port1[{$25}] : byte;  {$25}
Timer4[{$34}] : char; {$34}
Per2[{$28}] : word;  {$28}
```

{\$DATA}

**var**

```
x : byte;      {$0B}
y : char;     {$0C}
```

{\$EEPROM}

**var**

```
ex : byte;     {$0}
ey : char;     {$1}
```

## 7.2 External Memory

If external memory exists and in use then this memory must be declared and defined by the compiler switches {\$XDATA} {\$XDATA1} {\$XDATA2} {\$XDATA3} and {\$XDATA4} .

The definition of a XDATA area enables the use of the defined external memory area. The possibility to split the external RAM into max. 5 parts supports the implementation of external peripherals and also battery buffered RAM.

Normally the compiler initializes the entire memory with "0". In the case of peripherals and buffered RAM this is not desirable. Therefore it is possible to append the attribute **NoInit** to the definition. Then the initialization of this area is disabled.

**Define** *Xdata = StartAddr, EndAddr [, NoInit];*

If external memory is in use and defined, at first the area **XDATA** must be **defined**. Following further external areas can be defined with **XDATA1** etc. The definition includes the start- and end address of the memory block, followed by the optional **NoInit**.

```
Define ProcClock = 4000000;           {4 Mhz }
          SYSTICK  = 10;                {alle 10msec }
          XData    = $1000, $4FFF;      {16kB, start = 1000h, end = 4FFFh}
          XData1   = $8000, $80ff, NoInit; {Peripherie, no Init}
          StackSize = 82, iData;        {82 Bytes in iData}
          FrameSize = 99, xData;        {99 Bytes in xData}
```

```
var
{$XDATA}
  abc          : integer;
  fix[$3000]   : byte;
```

```
{XDATA1}
  port1[$8000] : byte;
  port2[$8001] : byte;
```

```
{IDATA}
```

System data, which can reside in an external area, for example StackSize or FrameSize, can only be allocated in the XDATA area.

## **{XIO +} {XIO -}**

Can be used in the XDATA area. It can tell the Optimizer about a memory mapped IO-area in XDATA. The memory area between these two switches then is treated in a different way by the optimizer. For example redundant accesses are not removed.

## 7.3 Include Files

### **{I Filename.ext}**

Reads an include-file, whereby "filename" also can contain a path. If there is no path it is searched within the working directory of the application. So sources, which are used again and again, can be included. The include-file can contain assembler or Pascal-source as well as both together. The usual conventions are further valid.

### **{J Filename.ext}**

Reads an include-file, whereby "filename" may not contain a path. The "Home-Directory" of the compiler is generally prefixed as path. This is very advantageous in case of procedures etc., which return again and again and it replaces almost a linker in the unit-concept.

### 7.3.1 Search Path for Include Files

Include files are searched in the following order:

1. actual working/project directory
2. in directories defined in Project Admin (IDE)
3. in directories defined in System Admin (IDE)
4. in the AVRco directory
5. in the "System" directory below the AVRco directory

## 7.4 Runtime Checks

If there is enough memory, "StackSize" can be enlarged. If the compiler now reports a stack overflow only the elimination of variables can help. This may be possible by removing unused variables. If there are no unused variables, it must be solved by a double-assignment of variables. But the programmer has to be extremely careful, to avoid a time based conflict of the assignment.

```
Var ch1          : byte;
    ch2[@ch1]   : byte;    {ch1 and ch2 same Address}
```

Please pay attention to chapter **RunTimeErr**.

### **{\$ShowError err: string}**

raises an error message

*{\$ShowError 'Error with ...}* raises an error message at assembly time

### **{\$ShowWarning err: string}**

raises a warning

*{\$ShowWarning 'Attention: ...}* raises a warning at assembly time

## 7.5 Variable, Constant and Procedure Check

### **{\$WG}**

Globally enables the check of variables and procedures. With obviously unused variables, ROM-constants or procedures the compiler produces a warning, which is evaluated by the IDE. This switch is off by default and can only be set to on.

### **{\$W+} {\$W-}**

Switches the check of variables and procedures on for the actual module (Main/Unit). With obviously unused variables, ROM-constants or procedures the compiler produces a warning, which is evaluated by the IDE.

(see also optimization)

Default: {\$W-}

### **{\$NORETURNCHECK}**

Only valid in conjunction with functions. With the following function there is no error message if there is no Return statement.

### **{\$NOOVRCHECK}**

Disables the Check for the following Variable Overlay Declaration.

### **{\$OverLay @VarName[, NoOvrCheck]} {\$OverLay 0}**

To avoid the separate overlay of each variable with: *yyy[@xxx] : byte;*

To place more than one variable into the referenced var, e.g. an existing Array or buffer.

"VarName" is must be existing variable (@VarName)in the RAM area.

The optional parameter "NoOvrCheck" defines that no overflow checks must take place. If this switch has a zero parameter "0" then this is the end of this overlay area.

All variables which are defined between these two switches now get upcounting addresses starting with the address of "VarName". This means they are placed "over" the basic variable.

Basically this variable at least should occupy the same or more memory as the sum of all overlaying ones. If an overflow is created it is displayed with the trailing switch. But this overflow is ignored with if the option "NoOvrCheck" is set with the leading switch.



Instead of a variable as the reference also an absolute address can be defined by

## **{\$OverLay \$nnnn, NoOvrCheck}**

Then the option "*NoOvrCheck*" is mandatory.

## **{\$Q-}**

disables the need to qualify identifiers within assembler coding.

Only for already existing older programs.

## **{\$TYPEDCONST OFF} {\$TYPEDCONST ON}**

For a better readability of programs and to avoid compiler errors.

With "ON" the compiler expects with each constant declaration also a proper type declaration. Because of this a "0" for example becomes unambiguous a byte, word, integer or a float.

```
const bb : byte = 0;
```

The switch is default "ON".

If existing programs should not be changed and to avoid error messages the typed const feature must be disabled in the main file like this:

```
{$TYPEDCONST OFF}
```

```
program ProgName;
```

## **{\$VALIDATE name}**

Constants and variables of the system and also of the application in most cases will be removed by the compiler/optimizer if they don't appear in the context. This means they must be included at least once in a statement.

This can lead to problems with inline assembler code in the Pascal sources.

The assembler shows an error because the called procedure etc. is not present in the asm source.

The switch **{\$VALIDATE name}** forces the compiler to import or implement the construct "name" and the optimizer is disabled for this item. The switch can only be used after the declaration of "name".

### **ATTENTION:**

this option (is the only one that) is not working with the "*Merlin Optimizer*"

## **{\$VALIDATE \$}**

Functions and procedures can be excluded from any optimiser remove with this switch. It must be placed before a function/procedure declaration.

## **{\$VALIDATE\_ALL}**

It tells the compiler to place obviously not used constants of type String, Array and Record into the program code. The optimization for these type of constants is then disabled completely.

## **{\$VALIDATE\_ON} {\$VALIDATE\_OFF}**

To set variable blocks as "used" so the message "possibly unused variable" becomes suppressed for this area.

## **{\$ZeroLocVars +}**

If active all local variables in functions and procedures are cleared to zero if a function or procedure is entered.

## **{\$NOADDRCHECK}**

For Mega128..256: Placing of constants always use the highest Flash Page as the destination. Basically other pages are not usable for this by the system. With this compiler switch the address or page check is disabled for the following constant definition. This constant can also be an external binary file which should be placed here. If the address given is not inside the uppermost flashpage (standard constant page), so an access to this constant by the application with its standard addressing mechanism is impossible. Then the application must provide very special own access methods.

## ***const***

```
{$NOADDRCHECK}
```

```
LookUpTab[$20000] : array[1..256] of byte = 'Name.ext'; //mega2561
```

## 7.6 System Controlling

### **{\$NOSAVE}**

Is only valid in connection with application interrupt procedures. With the following interrupt procedure the system does not save the working register automatically (except the status register and 4 main working registers). The programmer has to do it himself. It only makes sense for fast service - routines, written in assembler.

### **{\$NOREGSAVE}**

is only valid in connection with application interrupt procedures. With the following interrupt procedure the system does not save any the working registers. The programmer has to do it himself. It only makes sense for service routines written in assembler.

### **{\$NOSHADOW}**

The definition must be, if necessary, before the device declaration. With non-multitask applications only those working registers are saved, which are used by the interrupts within all interrupts. This saves essentially ram, rom, and run-time. This switch is overwritten by the import of tasks and processes.

### **{\$NOFRAME}**

Is only valid in connection with device driver procedures, which are called by the procedures **Write** or **Read**. With the following device procedure, which may only have one 8-bit passing parameter, this parameter is passed within a register. A parameter frame is not made. So local variables are not possible. This only makes sense for fast driver routines, written in assembler.

### **{\$DEBDELAY}**

For the simulator. Shortens the mDelays in the Simulators by 90%.  
This switch does not have any meanings for the generated Hexfile, i.e. it's not necessary to remove it.

### **{\$D+} {\$D-}**

Debug informations on or off. If off, the following statements are not stepped by the simulator.  
This switch does not have any meanings for the generated Hexfile, i.e. it's not necessary to remove it.

### **{\$X-} {\$X+}**

Disables the execution of a source code area in the Simulator. Useful if the program polls or waits for an external hardware.  
This switch does not have any meanings for the generated Hexfile, i.e. it's not necessary to remove it.

### **{\$DEVICE}**

This is only valid in connection with Device driver procedures, which are called by the procedures **Write** or **Read**. In the following Device procedure, which may only have an 8bit parameter; this parameter is passed in a register. A parameter frame is not built, so local variables are not possible. It only makes sense for fast driver routines written in assembler. Identical with the \$NOFRAME switch.

### **{\$LCDNOWAIT}**

Disables the Busy-Polling of the display driver. Only for debug purpose!

### **{\$LCDNOINIT}**

The LCD controller is not initialized at Reset time. The application must do it with a invocation of the system procedure "*LCDsetup*".

### **{\$ENUMTOASM}**

Enumerations normally are not exported into the assembler file to save computing time with the compiler and assembler and to have a good overlook over these files.

If the enum values are needed for inline assembly it is possible to force the export of the values of each enumeration into the asm-file as constants with this compiler switch.

## **{SSL+} {SSL-} (\*P\*)**

Enabled or disables the Smart-Linker. Using this switch removing dead code can be enabled (+) or disabled (-). These switches can be placed anywhere and without limitation Please note that this switch is always set to "off" at the entry point of an unit. The same is true for the main program file.

## **{SSL ON} (\*P\*)**

Switches the default mode of the linker at program start to active.

If the switch state is "on" (remove), it can be temporarily set "off" for a specific function or procedure by using the switch `{VALIDATE ProcedureName}`.

But this switch is only valid at locations where the concerned function is already known (defined) by the system. For validating before a function/procedure declaration use `{VALIDATE $}`.

## **{PCU} (\*P\*)**

The switch located in the IDE "Project/Project Options" controls all Units of the current project. If activated all Units of the project are processed by the precompiler and PCU files are build, dependant of the other "PCU copy" switches.

By the usage of this compiler switch in the source area of an Unit the rebuild of a PCU out of this Unit is forced, regardless of the meaning of the global switch in the IDE. But the "copy PCU" switches of the IDE are still valid and control the destination of this PCU.

## **{VectTab \$nnnn} (\*4\*)**

## **{CodeStart \$nnnn} (\*4\*)**

With standard AVR applications the Interrupt Vector Table always resides on address \$0000 in the code area (Flash). The same also is true with the Codestart. This means that also the generated code starts with address \$0000, but more exactly immediately behind the vector table. With these two switches the system can be forced to start the address generation with other addresses as the default ones.

The address parameter for these switches are always word addresses!

## **{BootApplication \$nnnn} (\*4\*)**

This switch is a combination the above two but in addition enables this application to use the Flash Downloader. So now it is possible to build an application which completely runs in the boot sections of the controller and still has access to all of the system resources and drivers because they are also placed into the boot section. The address parameter of this switche is always a word address and must be placed below the Device declaration!

A sample application can be found in the Demos Directory in "BootApp".

## 7.7 Conditional Compile

Sometimes it is necessary to generate different, for example hardware dependant versions of an application. The respective behavior can be controlled with the help of the compiler switch for depended compilation (Conditional Compile). The used "**Label**" only has a symbolic character. If the result of a switch is "false", so the here beginning source-code is treated as a comment as it does not exist until the switch is "true". All switches of this group can appear at any position within the source. An "IFxx"-introduction must be finished with an "ENDIF". There can be an "ELSE" in between. Nested IF's are permitted!

It is also possible to pass one or several DEFINES from the IDE PED32 to the compiler by using the menu item **Project/Project Options** and define a Label (without \$, parenthesis and DEFINE). The parameters and labels must be separated with a semicolon. These parameters are then treated by the compiler as if they are in the first source line and have the outline `{$DEFINE Label}`.

<code>{\$DEFINE label}</code>	sets "label" to true
<code>{\$UNDEF label}</code>	sets "label" to false
<code>{\$IFDEF label}</code>	If "label" is true, the following source is compiled upto "ELSE" or "ENDIF". If "label" is false, it is processed reverse.
<code>{\$ELSIFDEF label}</code>	If "label" is true, the following source is compiled.
<code>{\$IFNDEF label}</code>	If "label" is false, the following source is compiled upto "ELSE" or "ENDIF". If "label" is true, it is processed reverse.
<code>{\$ELSE }</code>	Reverses the actual state. If, for example the preceding source was compiled, so the following source is treated as a comment upto "ENDIF".
<code>{\$ENDIF}</code>	Closes an conditional block.

It's also possible to use a boolean expression consisting of DEFINES. Only the operators "AND" and "OR" are implemented and as arguments only previously introduced with `{$DEFINE ..}` can be used.

```
{$IFDEF ABC AND XYZ}
...
{$ELSIFDEF HIJK OR OPQ}
...
{$ENDIF}
```

Also parenthesis can be used:

```
{$IFNDEF ABC and (UVW or XYZ)}
...
{$ENDIF}
```

Further it is possible to check with IFDEF etc whether a Unit is present or not. Unit names and Unit filenames are included in the Define pool.

If the Unit-filename is "ABC.pas" and the internal Unit name is "U245" then the following is possible:

```
{$IFDEF FILE_ABC}           is true because the Unit filename is "ABC".
{$IFDEF U245}                is true because the internal Unit name is "U245".
```

```
Program Test;
{$DEFINE pp}
{$IFDEF pp}
```

```
Procedure ABC; {Procedure head}  
begin           {will be compiled and executed}  
  ...  
end;
```

```
{ELSE}  
Procedure ABC; {Procedure head}  
begin           {will be treated as comment}  
  ...  
end;  
{ENDIF}
```

## **{IF equation = true} {ELSIF equation = true}**

In place of values defined by \$DEFINE it's possible to use constants defined by the system or in the program source:

### **const**

```
  x : 25;  
  y : 1;
```

```
{IF x > y}
```

```
...
```

```
{ELSIF x = y}
```

```
...
```

```
{ENDIF}
```

```
{IF PROCCLOCK = 800000}
```

Examples of System defined constants for the Compiler Switches (and also for Statements):

```
_iDataStart  
_iDataEnd  
_EEPROMStart  
_EEPROMEnd  
_FlashStart  
_FlashEnd
```

## **{IFDEF CPUname}**

This switch makes it possible to build CPU type dependent code.

## **{HEXPATH 'pathname'}**

If "conditional compile" switches generate different firmware out of the same source, it makes sense to place the generated hex-files also into different directories. This compiler switch serves this purpose.

All involved tools which are called from the IDE like Editor, Compiler, Assembler, Programmer recognize this switch.

The argument 'pathname' must be delimited like any string in the system.

If the path or directory doesn't exist they will be created by the system.

## **{HEXNAME 'filename'}**

If "conditional compile" switches generate different firmware out of the same source, it makes sense to save the generated hex-files also with different file names. This compiler switch serves this purpose.

All involved tools which are called from the IDE like Editor, Compiler, Assembler, Programmer recognize this switch.

The argument 'filename' must be delimited like any string in the system.

Both switches can also be combined.

### **Note:**

A copy of the Flash and EEPROM hex files is also stored in the current project directory with their origin names so that the incircuit programmer at this point does not handle different locations for the same project. On the other hand the programmer's project administration must be extended with the different possible projects.



## 8 Program Structure

### 8.1 Program Frame

Because of syntax requirements a certain program frame is necessary. This starts with 'Program *name*' and ends with 'End.' As mentioned above in different chapters, diverse devices, system functions etc. have to be **imported** to be **defined**.

The programmer should take the example below as model for his own program structure. If there is a new project, there is automatically a main-file with the corresponding entries created by the IDE PED32. But the assumption is, that a template-file (\*.tmpl) exists and the entry of this file in the according "Control" of PED32.

Within some versions there is a so called **Application Wizard**. This is helpful to interactively create the source of a new application.

Of course the compiler creates a syntax error if there are false or missing imports or defines.

Unnecessary imports or variable declarations should be avoided. Think about the small resources (ram) of most processors. At least with the overwrite of variables and stacks the compiler gives a warning.

Nested procedures, long arithmetic statements with many brackets or procedure calls within Interrupts leads to run-time problems with the **stack-overflow** (parameter stack).

#### 8.1.1 Order

The order of the declarations of *program* up to *implementation* **must be** followed. *Const* and *Var* declarations can be mixed after that. **After the first procedure or function** declaration it is best to avoid to use global *Var* or *Const* declarations **as far as possible**.

**Local** variables or constants within a procedure or function mean a bigger code and a longer run-time. With procedures or functions, where as short process time as possible is required, it only should be coded with global variables. That is also true for passing parameters.

With **String** and **Array** variables you should be thrifty. The memory consumed can exhaust the system sources very fast. It is the same with the string conversions.

## 8.2 Initializing

After a reset or a program start the complete memory (variables) is set to zero (`var:= 0;`). So global variables need not be preinitialized by the program in general.

The initializing is directly after the introduction *implementation* or after calling the procedure **System\_Init**, if it exists.

Under some circumstances it could be necessary that some memory location are not initialized or erased. This can be achieved by the compiler switch **{\$NOINIT}**. But it must be clear that no more initialization is done after this switch. This means that all memory locations residing before this switch, are erased and all defined after this switch are unchanged.

The compiler and its system library only does such hardware initialization that is necessary for the function of the imported functions. If, for example **SwitchPort1** is imported, the defined port is switched to input. The import **ADCport** initializes all hardware, which is necessary for the execution of the converter. Separate bits (for example bit7 of LCDport) are not initialized and changed.

So the programmer **himself** has to initialize the hardware which he requires and which was **not imported**. The IO-Ports, which are in the CPU, are defined by the processor description-file `xxx.dsc` and imported by the compiler. So a new definition of the port by the programmer is not necessary. Example for PortB input/output mixed:

```
var
  DDRB[@PortB -1] : byte; { Data Direction reg PortB}
  Led1[@PortB, 0]  : bit;
  Rel1[@PortB, 1]  : bit;

begin
  DDRB= $0F;      {upper 4 bits Input, lower 4 bits output}
  EnableInts;     {Interrupt enable if necessary}
  Incl (Led1);    {Led1 On}
  Loop
    Toggle (Rel1); {switch Relais1}
    mDelay(1000); {1sec delay}
  endLoop;
end.
```



# AVRco Compiler-Manual

## Example of a Program Scheme:

**Program Test;**

**Device ...** {Hardware declaration}

**Import ...** {System functions/HardWare}

**From System Import ..** {System types/Software}

**Define ...** {Hardware Definition}

**Implementation ...** {Program start}

**Type ...** {Type declaration}

**Const ...** {Constant declaration}

**Var ...** {Variable declaration}

**Procedure System\_Init;** {optional}  
**begin**  
...  
**end;**

**Procedure ABC;** {Procedure head}  
**begin**  
...  
**end;**

**Function CDE : boolean;** {Function head}  
**begin**  
...  
**Return(a > b);** {Result of the function}  
**end;**

**Process PPP (20, 10 : iData);** {Process head}  
**begin**  
...  
**end;**

**Task TTTT (iData);** {Task head}  
**begin**  
...  
**end;**

**begin** {Main program start}  
...  
**EnableInts;**  
{Start\_Processes;} {if processes imported}  
**Loop**  
...  
**ABC;**  
**x:= CDE;**  
**EndLoop;**  
**end.**



## 9 Compiler Errors

### 9.1 Error File

If the compiler notices any errors it generates an error-file with the File Extension 'xxx.err'. This file is used by the IDE PED32 to locate errors and their display.

#### 9.1.1 Type Mismatch

Programming beginners will often stumble over the error **Type Mismatch**. The experienced programmer would be also astonished by the same fact, because maybe he knows and uses the automatic type conversion from Turbo Pascal, Delphi or C.

The automatic type-conversion, which is almost not existent within the compiler, gives the above mentioned error message.

There are several reasons, why this type of conversion does not exist in this compiler:

1. It is difficult and complex to realize a faultless automatic conversion within a compiler.
2. Every Turbo-programmer can sing a song about how to search for the cause of a wrong calculation result, to recognize finally, after many inspections of nested statements, that the type-conversion was the reason. To save the honor of Turbo it must be said that the formal compiler always behaves correctly, but the programmer often thinks in a different way about it. Often there is missing just a bracket pair.
3. With certain security relevant applications this kind of automatic conversion is not desirable.
3. It is the better and safer programming style if the programmer is forced to explain the compiler, what he wants. Or better: the programmer knows best, what should happen and how it should happen.
5. Type Casting (type conversion) is clearer and more readable `word := word (byte);`

A little problem should not kept secret at this point. It might appear that with constants < 256 and with certain operations, that the compiler does not know if it should treat the constant as an 8-bit or a 16-bit value. This might lead to a Type Mismatch.

```
var b : boolean;  
    i : integer;  
  
    b:= 5 > i;
```

When the compiler reaches the position '5' it only knows, that a boolean is expected and '5' can be a byte. So the '5' is treated as a byte and leads with the following integer 'i' to a Type Mismatch.

Remedy:

```
    b:= integer(5) > i;  
or  
    b:= i < 5;
```

## 10 Units (\*P\*)

### 10.1 Declaration and Construction of a Unit

A Unit consists of types, constants, variables and routines (procedures and functions). Each Unit has to be defined in a separate Unit-file (.PAS).

A Unit-file begins with the Unit-header and contains the sections **interface**, **implementation** (and optional **initialization**). The structure of a Unit-file looks like this:

*unit* Unit1;

**interface**

*uses* { List of additional Units }

{ interface-section }

**implementation**

{ implementation-section }

**initialization**

{ initialization-section }

**finalization**

{ finalization-section }

**end.**

A Unit must conclude with the word **end** followed by a period.

#### 10.1.1 Unit-Header

The Unit-header defines the name of the Unit. It consists of the reserved word **unit**, a valid identifier and a semicolon. The identifier must be the same name as the Unit-filename.

*unit* Hello;

This Unit-header can be used in a source file named Hello.PAS. The file with the compiled Unit then has the name Hello.PCU.

Unit-names must be unique in a project. Also when the Unit-files reside in different directories it's not possible to use multiple Units with identical names in a project.

## 10.1.2 Interface-Section

The interface-section of a Unit begins with the reserved word **interface**. It ends with the begin of an implementation-section. The interface-section declares constants, types, variables, procedures and functions which are accessible for Clients. Clients are other Units or the project itself, which import this Unit by a uses-clause. Such identifiers are called public because the Client can access them as they were declared in the Client itself.

The interface-declaration of a procedure or a function only contains the header of the routine. The block of the procedure or function must be declared in the implementation-section only. Procedure- and function declarations in the interface-section conform to the normal forward-declarations, but the forward directive must not be written.

The interface-section also can have an uses-clause, which directly must be placed after the interface statement.

## 10.1.3 Implementation-Section

The implementation-section of an Unit begins with the reserved word **implementation** and ends with the begin of the optional initialization-section or – if there is no initialization-section – with the finalization section or the end of the Unit. The implementation-section defines procedures and functions, which were defined in the interface-section. Within the implementation-section order of definitions or call of these procedures and functions is arbitrary. The parameter lists of functions and procedures must be the same as in the declarations in the interface section.

Except of the definitions of the published procedures and functions the implementation –section can have additional declarations of constants, types, variables, procedures and functions, which are private (local) for the Unit. Clients don't see this objects and can't access them.

With *Define\_USR* it is possible to define constants which are visible and accessible from every point of the program and also from each Unit.

This Define should only be used if it is absolute necessary because it's not a good programming style. The better way is to place such globals into an Unit which resides in the last position of the Unit chain. Then the definitions are also visible from all other parts of the application

## 10.1.4 Initialization-Section

The initialization-section is optional. It begins with the reserved word **initialization** and ends with the finalization section or the end of the Unit. The initialization-section contains statements, which are processed at program start in the given order. If you have to initialize hardware or variables concerning the Unit, it can be done here before the main program is invoked.

The initialization-sections of Units, which are imported by Clients, are processed in a hierarchical order. This means: the last Unit, which resides at the end of the chain is the first which is initialized.

## 10.1.5 Finalization-Section

The Finalization-Section is optional. It begins with the reserved word **finalization** and ends with the end of the Unit.

The statements in this block are processed when the application calls the system procedure "*System\_ShutDown*". The calling order of the finalizations is exactly opposite to the calling order of the initializations. The procedure "*System\_ShutDown*" simply calls the Finalization Statements and nothing else. The "Finalization" is useful for a dedicated run down of the system before a switch off or a start of the sleep mode



# AVRco Compiler-Manual

## 10.1.6 Uses-Clause

The uses-clause of the main-program defines all Units, which are imported into the program. These Units themselves also can have own uses-clauses. A uses-clause in a program or a Unit defines the Units which are used by this module. A uses-clause can be used in the source of the following files:

Mainfile of a program  
Interface-section of a Unit

The Unit **System** is automatically imported by an application and must not be imported in a uses-clause. (The Unit System implements routines for hardware and software drivers, string conversion, Floating point etc.).

A uses-clause always consists of the reserved word **uses**, one or several unit names, separated by commas. It concludes with semicolon.

*uses Hello, MyMath;*

In the uses-clause of a program it's possible that each Unit name can be extended by the reserved word **in** followed by the path and name of the source file. The name is written with or without the path in quotes. The path can be absolute or relative.

*uses Hello, MyMath in 'C:\MyProgMyMath', InitUnit;*

Use **in ...** after the unit name, if you must define the filename of a Unit source. The reserved word **in** is only necessary if the location of the source is ambiguous for the following reason:

the Unit-source file is in a different location than the project itself, and this directory is not in the search path of the compiler nor in the home path of the compiler.

### 10.1.6.1 Search Path for Units

Units are searched in the following order:

1. actual working/project directory
2. in directories defined in Project Admin (IDE)
3. in directories defined in System Admin (IDE)
4. in the AVRco directory
5. in the "System" directory below the AVRco directory  
Precompiled Units (PCUs) initially included in the AVRco system are placed here

## 10.1.7 Info Part of a Unit

The Info part of an Unit consists of any lines at any positions in this Unit.  
An Info line must start with

*/// and here the info*

The three pipe chars must not contain spaces. The system treats an Info line like a comment.

In the IDE (Editor) there is a menu item "Project/Unit infos". All Units which are declared in the SourceCodeControlSystem (SCCS) are listed here.  
A mouseclick onto an Unit name opens a window which shows the Info part of this Unit

## 10.1.8 Hardware Imports within Units

If Units are imported the definitions can also be placed into an Unit:

### Main Program

```
Import SysTick, MatrixPort, SerPort;
```

```
From System Import longword, longint, float, pipes;
```

### **Define**

```
ProcClock = 8000000; {Hertz}  
SysTick   = 10;      {msec}  
StackSize = $0020, iData;  
FrameSize = $0040, iData;  
SerPort   = 9600;  
RxBuffer  = 16, iData;
```

```
DefineFrom unit1; // Unit1 defines the Matrixport
```

### Unit

```
Unit Unit1;
```

### **Define**

```
MatrixRow = PortD, 4; {use PortD, start with bit4}  
MatrixCol = PinD, 0;  {use PinD, start with bit0}  
MatrixType = 3, 4;    {3 Rows at PortD, 4 Columns at PinD}
```

### **Interface**

...

The reserved word "*DefineFrom*" within the main "Define" block switches the scanning from main program to the given Unit name, where the scanning of the defines continues.

When the word "*Interface*" appears the scanning is switched back to the main program.

## 10.2 PreCompiled Units

The Profi-version allows to build precompiled units and also include files. This works similar like a linker, but isn't really exact the same.

The units will be precompiled and therefore are not readable for others. But in the compiled project one can see the generated asm sources of these units. Normally this isn't a problem at all, because the generated asm is also visible in the asm window of all debuggers and simulators.

In the IDE PED32 in the menu "Project/Project options" there is a checkbox "Precompile Units". If this box is checked, all units and includes are precompiled into files with the extension "filename.PCU".

These files can be passed to others without their accompanying sources.

## 11 Assembler

### 11.1 Overview

The AVRco System includes an Assembler, which can be used as a stand-alone program. Please note that all processor mnemonics can be used, but no macros and similar function are possible like complex assembler have.

Basically the assembler is used for assembly of the asm sources generated by the compiler and also for asm statements defined with the "ASM"- definitions in the Pascal-Source.

The construction and meanings of the mnemonics can be found in the processor manuals. With some processors it's possible that some mnemonics, e.g. register names, can collide with variable names of the application. Then other names are used for that definition.

**ASM: mnemonic** ;Definition of a single assembler statement.

#### 11.1.1 ASM;

Start of an assembler text block

A program for Embedded Control does not often work without an assembler code, because either the compiler generated code is too slow for some operations, or certain assembler commands have to be done, which are not known or not used by the compiler.

It is possible to include the assembler source directly at every position of the Pascal source. This source is passed non-tested and unmodified by the compiler to the assembler. Because the compiler also generates assembler code, the assembler text is seamless inserted.

Asm-syntax errors are only recognized by the assembler, not by the compiler. All declared variables are accessible in the assembler text.

#### **Warning:**

Labels in a assembler-block have to start at the beginning of a line and limited with a ':'.  
This line may not contain any further instructions, e.g. code. The analysis of the compiler generated assembler files 'xxx.ASM' may help.

```
ASM;  
  LDI      _ACCA, 67;  
  STS     a, _ACCA;      {a = Pascal var }  
ENDASM;
```

#### 11.1.2 ENDASM;

End of an assembler text

ASM and ENDASM are Pascal statements and must end with a semicolon ;  
Exception: single assembler statement with **ASM:**

## 11.2 Assembler - Keywords

### 11.2.1 Register

With the AVR the register names 'R0'.. 'R31' in the mnemonics are changed to `_ACCA`, `_ACCB` etc. in order to avoid naming conflicts with possible Pascal vars or constants.

Relations between register numbering and pseudo ACCUs:

R0	=	<code>_ACCGLO</code>	Arithmetic reg and Flash access
R1	=	<code>_ACCGHI</code>	only with Imports of 32bit Types and Floats
R2	=	<code>_ACCHLO</code>	only with Imports of 32bit Types and Floats
R3	=	<code>_ACCHHI</code>	only with Imports of 32bit Types and Floats
R4, R5	=	<code>\$_CURPROCESS</code>	no User Access if defined by the system
R6, R7	=	<code>\$_CURTASK</code>	no User Access if defined by the system
R8, R9	=	<code>\$_SAVERET</code>	no User Access if defined by the system
R10	=	<code>FLAGS</code>	
R11	=	<code>FLAGS2</code>	
R12	=	<code>_SYSTFLAGS</code>	no User Access if defined by the system
R13	=		not used and not defined
R14	=		not used and not defined
R15	=		not used and not defined
R16	=	<code>_ACCB</code>	main working register low byte (16bit and 32 bit types)
R17	=	<code>_ACCA</code>	main working register 8bit types hi byte (16bit types) hi byte of low word (32bit and floats)
R18	=	<code>_ACCALO</code>	main working register Lo byte of hi word (32bit and float)
R19	=	<code>_ACCAHI</code>	main working register Hi byte of hi word (32bit and float)
R20	=	<code>_ACCDLO</code>	Arithmetic reg
R21	=	<code>_ACCDHI</code>	Arithmetic reg
R22	=	<code>_ACCELO</code>	Arithmetic reg
R23	=	<code>_ACCEHI</code>	Arithmetic reg
R24	=	<code>_ACCFLO</code>	Arithmetic reg
R25	=	<code>_ACCFHI</code>	Arithmetic reg
R26	=	<code>_ACCBLO</code>	Arithmetic reg
R27	=	<code>_ACCBHI</code>	Arithmetic reg
R26, R27	=	X-register	second pointer reg
R28, R29	=	<code>_FRAMEPTR</code>	
R30	=	<code>_ACCCLO</code>	Arithmetic reg
R31	=	<code>_ACCCHI</code>	Arithmetic reg
R30, R31	=	Z-register	main pointer reg

All the above registers `_ACCxx`, excluding those, which are only present with 32bit types, can be used freely within the assembler part of the source. Those registers which are named and imported with 32bit imports (`_ACCGHI`, `_ACCHHI`, `_ACCHLO`) can only be used if a 32bit type is imported by an Import definition (LongInt, LongWord or Float).

All other registers which are named by the Compiler (`$_CURPROCESS`, `FRAMEPTR` etc) should only be read, but never written or altered in any way, otherwise a system crash at runtime is certain.



# AVRco Compiler-Manual

An exception are the 3 register pairs X, Y and Z. These mnemonics respective register names must be used in conjunction with pointer operations.

```
LDI  _ACCCLO, 00h;
LDI  _ACCCHI, 10h;      load _ACCCLO/HI = Z-reg with 1000h
LD   _ACCA, Z+;        load _ACCA with contents of RAM loc 1000h
```

For better readability the registers R28..R30 have additional names:

```
R26 -> XL  LDI XL, 45h
R27 -> XH
R28 -> YL
R29 -> YH
R30 -> ZL
R31 -> ZH  LDI ZH, 01
```

## Labels

Within an assembler block labels must start immediately at the **line's start** and must end with a colon : . Appended directives or mnemonics after the colon lead to an assembler error. All other lines, except comments and definitions, must have at least one leading space.

## Definitions

must also start immediately at the **line's start** and must end with a assembler directive. No colon is used. The corresponding assembler directive must be on the same line.

## 11.2.2 Assembler Directives

### **.ORG addr**

changes the internal program-counter of the assembler to the address "addr". Should not be used by the programmer, it's reserved for the Compiler

### **name .EQU nn**

Constant definition. Can be used. The better way is to define constants in the Pascal Code. Then they are accessible in assembler and also in Pascal.

### **.BYTE .WORD .ASCII**

Constant placed into the Flash. Should not be used. With the mega it's forbidden, because here the constants are always placed into the second 64k-Page.

### **.END**

End of the assembler text. Is defined by the Compiler.

### **ADDI ADCI**

The AVR doesn't know the opcodes "ADDI" and "ADCI".

These can be easily implemented with SUBI/SBCI and a negated argument. But the carry-flag can't be used as previously. ADDI and ADCI now are implemented as pseudo-ops in the Assembler.

```
ADDI R16, nn
ADCI R16, nn
```



## ADIW SBIW

The mnemonics "ADIW" and "SBIW" in the origin Atmel way of writing are related to R26/X R28/Y and R30/Z. To improve the readability the Assembler is extended to accept the following mnemonics:

```
ADIW X, nn
ADIW Y, nn
ADIW Z, nn
SBIW X, nn
SBIW Y, nn
SBIW Z, nn
```

## SYSTEM.VectTab

For JUMPs and CALLs to the absolute program start (ResetVector 0) it is possible to use Assembler statements like

```
JMP $0000
CALL $0000
```

This absolute addressing mode is only possible with assembler and restricted to mega16/32/,...  
Smaller CPUs don't support absolute CALL/JMPs.

To use it also with these types and also in Pascal there is the call/jump target label "SYSTEM.VectTab"

```
RJMP SYSTEM.VectTab
RCALL SYSTEM.VectTab
JMP SYSTEM.VectTab
CALL SYSTEM.VectTab
```

## 11.2.3 Operators for Constant Manipulation

<b>NOT</b>	Inverts the following argument. <i>LDI _ACCA, NOT 0FFh;</i> <i>load _ACCA with 00h</i>
<b>AND</b>	logical AND of two parameters. <i>LDI _ACCA, 0FFh AND 0F0h;</i> <i>load _ACCA with 0F0h</i>
<b>SHRB</b>	Shift operation with a byte result. <i>LDI _ACCA, 0FF00h SHRB 8;</i> <i>load _ACCA with 0FFh</i>
<b>SHLB</b>	Shift operation with a byte result. <i>LDI _ACCA, 0FFh SHLB 4;</i> <i>load _ACCA with 0F0h</i>
<b>RORB</b>	Rotate with a byte result. <i>LDI _ACCA, 0A5h RORB 4;</i> <i>load _ACCA with 05Ah</i>
<b>ROLB</b>	Rotate with a byte result. <i>LDI _ACCA, 081h ROLB 1;</i> <i>load _ACCA with 03h</i>

The Assembler accept character constants, too.

```
LDI _ACCA, 'z';
```

## 11.2.4 Access to Pascal Constants and Variables

As stated above an assembler statement can access constants and vars which are declared in the Pascal source. Please note that a var normally must be qualified by it's **Modulename**. With accesses to words and longs bear in mind that the AVR uses the "Little Endian" principle. This means that the loByte of a word is determined by the lower order of the address, the hiByte by the higher order.

**var**

```
bb : byte;
ww : word;
```

**begin**

**ASM;**

```
LDS    _ACCA, module.bb    ; load byte bb to _ACCA
; load address of var ww to pointer reg Z
LDI    _ACCCLO, module.ww AND 0FFh;
LDI    _ACCCHI, module.ww SHR 8;
;
; the Z-reg contains now the adr of var ww
; store a 0FFh to loByte of ww
LDI    _ACCA, 0FFh;
ST     Z, _ACCA;
;
; store a 00h to hiByte of ww
LDI    _ACCA, 00h;
ST     Z+1, _ACCA;
;
; move hiByte of ww to bb
LDS    _ACCA, module.ww + 1;
STS    module.bb, _ACCA;
```

**ENDASM;**

**End;**

The basically necessary **Modulname** can be replaced by a %.

Example:

**Unit ABC;**

**var xyz : byte;**

...

**ASM**

```
LDS    _ACCA, ABC.xyz;
; or
LDS    _ACCA, %.xyz
```

;

## 11.3 Assembler Routines

### 11.3.1 Local Variables and Assembler Access

Local variables in procedures and functions, as well as the passed parameters, are stored temporarily in the Frame. The addressing of the Frames is via the FramePointer (Y-register). It's impossible to access these vars and parameters by their names, because these types are always located relative to the FramePointer and not absolute.

The FramePointer always points to the last defined variable/parameter. The offset of the previous respective the above parameters must be calculated by the assembler programmer himself. Here he must take in account the order of the definitions and also the memory representation in bytes of the used types.

An erroneous offset calculation can lead to a system crash at runtime, at least with a write access.

**Procedure** *LocTest* (*var x : byte; bb : byte; ww : word; pt : pointer*);

**var**

*Lbb* : *byte*;  
*Lww* : *word*;  
*Lpt* : *pointer*;

**begin**

**ASM;**

```

; Access to local pointer Lpt
; copy Lpt to Z-reg
LD  _ACCCLO, Y           ; lo byte of Lpt
LDD _ACCCHI, Y+1        ; Y-reg offset 1
;
; Access to local word Lww
; copy ww to _ACCA, _ACCB
LDD _ACCB, Y+2          ; lo byte of Lww
LDD _ACCA, Y+3          ; hi byte of Lww
;
; Access to local byte Lbb
; copy lobyte of Lww to Lbb
LDD _ACCA, Y+2          ; lo byte of Lww
STD  Y+4, _ACCA         ; byte Lbb
;
; Access to parameter pointer pt
; copy pt to Z-reg
LDD _ACCCLO, Y+5        ; lo byte of pt
LDD _ACCCHI, Y+6        ; hi byte of pt
;
; Access to parameter word ww
; copy ww to _ACCA, _ACCB
LDD _ACCB, Y+7          ; lo byte of ww
LDD _ACCA, Y+8          ; hi byte of ww
;
; Access to parameter byte bb
; copy lobyte of ww to bb
LDD _ACCA, Y+7          ; lo byte of ww
STD  Y+9, _ACCA         ; byte bb
;
; Access to parameter byte x
; mov a 00h to x
; remember that a var parameter is passed by it's address
; and not by it's value. So this param is always a pointer !!!
LDD _ACCCLO, Y+10       ; lo byte of adr of x
LDD _ACCCHI, Y+11       ; hi byte of adr of x
LDI _ACCA, 00h          ; load a zero
ST  Z, _ACCA           ; store it with Z-Pointer reg to x
;

```

**ENDASM;**

**end;**

Please note, that addressing with the use of the pointer registers of the AVR, in this case the Y-register (FramePointer) is only possible with an offset within 0..+63. If the Frame incl. Parameters is larger than 63 bytes, and the offset of the desired parameters/var is larger than 63, so the FramePointer Y must be copied into another pointer register (eg X or Z) and the offset must be added to this register pair as a constant. Never manipulate the Y-register !!



# AVRco Compiler-Manual

## 11.3.2 Procedure Calls and System Functions

Inside an assembler block it's possible to call user defined Pascal procedures/functions and also system functions. Please note that the required Jump and Call labels must be qualified. With system functions (ser. interface etc.) the qualifier "SYSTEM" must be used. With user functions in the main program the program name (not the filename) and with functions in Units the Unit name (not the filename) must be used

*Program CallTest;*

...

*Procedure isCalled;*

*begin*

...

*end;*

**ASM;**

*LDI        \_ACCA, 2Ah;        " \* "*

*RCALL     SYSTEM.SEROUT;*

*RCALL     CallTest.isCalled;*

**ENDASM;**

## 11.3.3 Function Results and Assembler

The results of a function are always passed in registers to the calling location.

### 8bit Results (Byte, Char, Boolean etc)

Register *\_ACCA*

### 16bit Results (Word, Integer, Pointer etc)

Register *\_ACCA* (hiByte) *\_ACCB* (loByte)

### 32bit Results (LongWord, LongInt, Float etc)

Register *\_ACCA* (hiByte, loWord) *\_ACCB* (loByte, loWord)

Register *\_ACCAHI* (hiByte, hiWord) *\_ACCALO* (loByte, hiWord)

## 11.3.4 Function/Procedure Exit

The Compiler often generates, depending on the construction of the procedure or function, exit code at the place of the Pascal End-Statement. Because of this there should be no *RET* or *RETI* statement programmed. The result can be a strange behavior of the procedure or system. The better way is to place an *ASM-Label* before the End-Statement and then place a *RJMP* into the code to get to the exit code of the compiler.

## 11.3.5 Interrupt Procedures with Assembler

The Compiler saves at Interrupts always the registers `_ACCA`, `_ACCB`, `_ACCLO` and `_ACCCHI`. If processes or Tasks are imported, before calling the interrupt procedure all registers are saved. This can be disabled by the compiler switch `{$NOSAVE}` direct before the interrupt procedure. An other switch is `{$NOREGSAVE}` which disables any register saving. For special purposes, (very dangerous).

If no processes or Tasks are imported, the global compiler switch `{$NOSHADOW}` disables the complete saving of the registers. Alternatively one can disable the complete saving with `{$NOSAVE}` and only the 4 Accus are saved.

With complete saving of the registers each ACCU can be used within the assembler code. Otherwise only the ACCUs `_ACCA`, `_ACCB`, `_ACCLO` and `_ACCCHI` can be used. If there is a need for more ACCUs, these must be saved by a `PUSH` and restored by a `POP`.

A complete register saving needs about 20 Bytes in RAM, statically, not on Stack or Frame. Because of this a stacked interrupt is impossible. That means within an interrupt procedure the interrupt never should be enabled again. The CPU itself does this enabling with the `RETI` instruction.

With the minimal saving of the 4 ACCUS (pushed onto the Stack) an Enable Interrupt is possible, but is strongly discouraged. The results can be catastrophic, depending of the operations within the procedure.

## 11.3.6 Constants and Optimization

Constants, variables and system functions which are only accessed within an assembler block are normally removed by the optimizer. Because the Compiler doesn't analyze the assembler code it doesn't have any access to these parts. Therefore there can be errors while assembling the project, because the referenced labels don't exist.

In order to avoid optimization removing such constructs (e.g. constants) it is necessary to make at least one access in HLL (Pascal) of these constants etc.

## 11.4 Assembler Switches

The assembler knows several command-line switches, which can be specified with the name of the source-file by a calling within the Batch-Mode by PED32. Because this tool needs many additional infos from the system it is impossible to run the assembler outside of the IDE PED32.

```
-R   Assembler runs without showing itself in the background
-H   Hexfile output with the FileExtension 'xxx.hex'
-L   Listfile output on. FileExtension      'xxx.lst'
```

## 11.5 Assembler Errors

If the assembler recognizes any errors then it generates an error-file with the FileExtension 'xxx.err'. This file is used by the IDE PED32 to locate and display the error.

Assembler errors should generally not appear, unless assembler statements are used.

---

©1996-2008 ***E-LAB Computers***  
Grombacherstr. 27  
D74906 Bad Rappenau

Tel. 07268/9124-0  
Fax. 07268/9124-24

Internet: [www.e-lab.de](http://www.e-lab.de)  
e-mail: [info@e-lab.de](mailto:info@e-lab.de)

---