

AVRTools

Generated by Doxygen 1.9.8

1 AVRTools: A Library for the AVR ATmega328 and ATmega2560 Microcontrollers	1
1.1 Overview	1
1.2 Audience	2
1.3 AVRTools is not...	2
1.4 Quick Tour of AVRTools	3
1.4.1 Foundational Elements and Concepts	3
1.4.2 What you need to know about pin name macros	3
1.4.3 The core modules	4
1.4.3.1 System initialization module	5
1.4.3.2 System clock module	5
1.4.3.3 Analog-to-Digital module	5
1.4.3.4 PWM module	6
1.4.3.5 Minimal USART modules	6
1.4.3.6 ABI module	6
1.4.3.7 New module	6
1.5 Sample start up code using AVRTools	7
1.6 Advanced modules	7
1.7 Documentation	7
1.8 Questions	7
2 Advanced Features	9
2.1 Advanced serial (USART) module	9
2.2 I2C modules	10
2.2.1 I2C Master module	11
2.2.2 I2C Slave module	11
2.3 I2C-based LCD module	12
2.4 Interrupt utilities module	12
2.5 SPI module	12
2.6 Memory utilities module	14
2.7 Simple delays module	14
2.8 GPIO pin variables	14
2.8.1 Example using GPIO pin macros	16
2.8.2 Example using GPIO pin variables	16
3 FAQ	19
3.1 Frequently Asked Questions	19
3.2 Can AVRTools be installed as an Arduino IDE Library?	19
3.3 Why can't I assign pins like pPin01 to a variable?	19
3.4 Why isn't the SPI module asynchronous?	20
3.5 Why does the SPI module only implement master mode?	20

3.6 Why is there a setGpioPinHigh() macro and a _setGpioPinHigh() macro?	20
3.7 _setGpioPinHigh() is defined with 8 arguments, but called with 1 argument—how can that work?	20
3.8 Why is there a setGpioPinHigh() macro and a setGpioPinHighV() function?	21
4 Namespace Index	23
4.1 Namespace List	23
5 Hierarchical Index	25
5.1 Class Hierarchy	25
6 Class Index	27
6.1 Class List	27
7 File Index	29
7.1 File List	29
8 Namespace Documentation	33
8.1 I2cMaster Namespace Reference	33
8.1.1 Detailed Description	35
8.1.2 Enumeration Type Documentation	35
8.1.2.1 I2cBusSpeed	35
8.1.2.2 I2cPullups	36
8.1.2.3 I2cSendErrorCodes	36
8.1.2.4 I2cStatusCodes	36
8.1.3 Function Documentation	37
8.1.3.1 busy()	37
8.1.3.2 pullups()	37
8.1.3.3 readAsync() [1/2]	37
8.1.3.4 readAsync() [2/2]	38
8.1.3.5 readSync() [1/2]	38
8.1.3.6 readSync() [2/2]	39
8.1.3.7 start()	39
8.1.3.8 stop()	39
8.1.3.9 writeAsync() [1/4]	40
8.1.3.10 writeAsync() [2/4]	40
8.1.3.11 writeAsync() [3/4]	41
8.1.3.12 writeAsync() [4/4]	42
8.1.3.13 writeSync() [1/4]	42
8.1.3.14 writeSync() [2/4]	43
8.1.3.15 writeSync() [3/4]	43
8.1.3.16 writeSync() [4/4]	44

8.2 I2cSlave Namespace Reference	44
8.2.1 Detailed Description	45
8.2.2 Enumeration Type Documentation	45
8.2.2.1 I2cBusSpeed	45
8.2.2.2 I2cPullups	45
8.2.2.3 I2cStatusCodes	46
8.2.3 Function Documentation	46
8.2.3.1 busy()	46
8.2.3.2 processI2cMessage()	46
8.2.3.3 pullups()	47
8.2.3.4 start()	47
8.2.3.5 stop()	48
8.3 Interrupts Namespace Reference	48
8.3.1 Detailed Description	48
8.3.2 Enumeration Type Documentation	48
8.3.2.1 ExternalInterrupts	48
8.3.2.2 PinChangeInterrupts	49
8.4 MemUtils Namespace Reference	49
8.4.1 Detailed Description	50
8.4.2 Function Documentation	50
8.4.2.1 freeMemoryBetweenHeapAndStack()	50
8.4.2.2 freeSRAM()	50
8.4.2.3 getFreeListStats()	50
8.4.2.4 memoryAvailableOnFreeList()	51
8.4.2.5 resetHeap()	51
8.5 SPI Namespace Reference	51
8.5.1 Detailed Description	52
8.5.2 Enumeration Type Documentation	53
8.5.2.1 ByteOrder	53
8.5.2.2 SpiMode	53
8.5.3 Function Documentation	54
8.5.3.1 configure()	54
8.5.3.2 disable()	54
8.5.3.3 enable()	55
8.5.3.4 transmit() [1/2]	55
8.5.3.5 transmit() [2/2]	55
8.5.3.6 transmit16()	56
8.5.3.7 transmit32()	56
8.6 USART0 Namespace Reference	56

8.6.1 Detailed Description	57
8.6.2 Function Documentation	57
8.6.2.1 available()	57
8.6.2.2 flush()	57
8.6.2.3 peek()	58
8.6.2.4 read()	58
8.6.2.5 start()	58
8.6.2.6 stop()	58
8.6.2.7 write() [1/4]	59
8.6.2.8 write() [2/4]	59
8.6.2.9 write() [3/4]	59
8.6.2.10 write() [4/4]	60
8.7 USART1 Namespace Reference	60
8.7.1 Detailed Description	61
8.7.2 Function Documentation	61
8.7.2.1 available()	61
8.7.2.2 flush()	61
8.7.2.3 peek()	61
8.7.2.4 read()	61
8.7.2.5 start()	62
8.7.2.6 stop()	62
8.7.2.7 write() [1/4]	62
8.7.2.8 write() [2/4]	63
8.7.2.9 write() [3/4]	63
8.7.2.10 write() [4/4]	64
8.8 USART2 Namespace Reference	64
8.8.1 Detailed Description	65
8.8.2 Function Documentation	65
8.8.2.1 available()	65
8.8.2.2 flush()	65
8.8.2.3 peek()	65
8.8.2.4 read()	65
8.8.2.5 start()	66
8.8.2.6 stop()	66
8.8.2.7 write() [1/4]	66
8.8.2.8 write() [2/4]	67
8.8.2.9 write() [3/4]	67
8.8.2.10 write() [4/4]	68
8.9 USART3 Namespace Reference	68

8.9.1 Detailed Description	69
8.9.2 Function Documentation	69
8.9.2.1 available()	69
8.9.2.2 flush()	69
8.9.2.3 peek()	69
8.9.2.4 read()	69
8.9.2.5 start()	70
8.9.2.6 stop()	70
8.9.2.7 write() [1/4]	70
8.9.2.8 write() [2/4]	71
8.9.2.9 write() [3/4]	71
8.9.2.10 write() [4/4]	71
9 Class Documentation	73
9.1 Interrupts::AllOff Class Reference	73
9.1.1 Detailed Description	73
9.2 Interrupts::ExternalOff Class Reference	73
9.2.1 Detailed Description	74
9.2.2 Constructor & Destructor Documentation	74
9.2.2.1 ExternalOff()	74
9.3 GpioPinVariable Class Reference	74
9.3.1 Detailed Description	75
9.4 I2cLcd Class Reference	76
9.4.1 Detailed Description	79
9.4.2 Member Enumeration Documentation	79
9.4.2.1 anonymous enum	79
9.4.2.2 anonymous enum	79
9.4.2.3 IntegerOutputBase	80
9.4.3 Member Function Documentation	80
9.4.3.1 command()	80
9.4.3.2 displayBottomRow()	80
9.4.3.3 displayTopRow()	81
9.4.3.4 init()	81
9.4.3.5 print() [1/10]	81
9.4.3.6 print() [2/10]	81
9.4.3.7 print() [3/10]	82
9.4.3.8 print() [4/10]	82
9.4.3.9 print() [5/10]	82
9.4.3.10 print() [6/10]	83

9.4.3.11 print() [7/10]	83
9.4.3.12 print() [8/10]	84
9.4.3.13 print() [9/10]	84
9.4.3.14 print() [10/10]	85
9.4.3.15 println() [1/10]	85
9.4.3.16 println() [2/10]	85
9.4.3.17 println() [3/10]	86
9.4.3.18 println() [4/10]	86
9.4.3.19 println() [5/10]	86
9.4.3.20 println() [6/10]	87
9.4.3.21 println() [7/10]	87
9.4.3.22 println() [8/10]	87
9.4.3.23 println() [9/10]	88
9.4.3.24 println() [10/10]	88
9.4.3.25 readButtons()	88
9.4.3.26 setBacklight()	89
9.4.3.27 setCursor()	89
9.4.3.28 write() [1/4]	89
9.4.3.29 write() [2/4]	90
9.4.3.30 write() [3/4]	90
9.4.3.31 write() [4/4]	90
9.5 Interrupts::PinChangeOff Class Reference	91
9.5.1 Detailed Description	91
9.5.2 Constructor & Destructor Documentation	91
9.5.2.1 PinChangeOff()	91
9.6 Reader Class Reference	92
9.6.1 Detailed Description	93
9.6.2 Member Function Documentation	93
9.6.2.1 available()	93
9.6.2.2 find() [1/2]	94
9.6.2.3 find() [2/2]	94
9.6.2.4 findUntil() [1/2]	94
9.6.2.5 findUntil() [2/2]	95
9.6.2.6 peek()	95
9.6.2.7 read()	95
9.6.2.8 readBytes() [1/2]	96
9.6.2.9 readBytes() [2/2]	96
9.6.2.10 readBytesUntil() [1/2]	96
9.6.2.11 readBytesUntil() [2/2]	97

9.6.2.12 readFloat() [1/2]	97
9.6.2.13 readFloat() [2/2]	97
9.6.2.14 readLine()	98
9.6.2.15 readLong() [1/2]	98
9.6.2.16 readLong() [2/2]	98
9.6.2.17 setTimeout()	99
9.7 RingBuffer Class Reference	99
9.7.1 Detailed Description	100
9.7.2 Constructor & Destructor Documentation	100
9.7.2.1 RingBuffer()	100
9.7.3 Member Function Documentation	100
9.7.3.1 isEmpty()	100
9.7.3.2 isFull()	100
9.7.3.3 isEmpty()	101
9.7.3.4 isNotFull()	101
9.7.3.5 peek()	101
9.7.3.6 pull()	101
9.7.3.7 push()	102
9.8 RingBufferT< T, N, SIZE > Class Template Reference	102
9.8.1 Detailed Description	103
9.8.2 Member Function Documentation	103
9.8.2.1 discardFromFront()	103
9.8.2.2 isEmpty()	103
9.8.2.3 isFull()	104
9.8.2.4 isEmpty()	104
9.8.2.5 isNotFull()	104
9.8.2.6 peek()	104
9.8.2.7 pull()	105
9.8.2.8 push()	105
9.9 Serial0 Class Reference	106
9.9.1 Detailed Description	109
9.9.2 Member Enumeration Documentation	109
9.9.2.1 IntegerOutputBase	109
9.9.3 Member Function Documentation	109
9.9.3.1 available()	109
9.9.3.2 find() [1/2]	110
9.9.3.3 find() [2/2]	110
9.9.3.4 findUntil() [1/2]	110
9.9.3.5 findUntil() [2/2]	111

9.9.3.6 peek()	111
9.9.3.7 print() [1/10]	111
9.9.3.8 print() [2/10]	112
9.9.3.9 print() [3/10]	112
9.9.3.10 print() [4/10]	112
9.9.3.11 print() [5/10]	113
9.9.3.12 print() [6/10]	113
9.9.3.13 print() [7/10]	114
9.9.3.14 print() [8/10]	114
9.9.3.15 print() [9/10]	115
9.9.3.16 print() [10/10]	115
9.9.3.17 println() [1/10]	115
9.9.3.18 println() [2/10]	116
9.9.3.19 println() [3/10]	116
9.9.3.20 println() [4/10]	116
9.9.3.21 println() [5/10]	117
9.9.3.22 println() [6/10]	117
9.9.3.23 println() [7/10]	117
9.9.3.24 println() [8/10]	118
9.9.3.25 println() [9/10]	118
9.9.3.26 println() [10/10]	118
9.9.3.27 read()	119
9.9.3.28 readBytes() [1/2]	119
9.9.3.29 readBytes() [2/2]	119
9.9.3.30 readBytesUntil() [1/2]	120
9.9.3.31 readBytesUntil() [2/2]	120
9.9.3.32 readFloat() [1/2]	121
9.9.3.33 readFloat() [2/2]	121
9.9.3.34 readLine()	121
9.9.3.35 readLong() [1/2]	122
9.9.3.36 readLong() [2/2]	122
9.9.3.37 setTimeout()	122
9.9.3.38 start()	123
9.9.3.39 stop()	123
9.9.3.40 write() [1/4]	123
9.9.3.41 write() [2/4]	124
9.9.3.42 write() [3/4]	124
9.9.3.43 write() [4/4]	124
9.10 Serial1 Class Reference	125

9.10.1 Detailed Description	128
9.10.2 Member Enumeration Documentation	128
9.10.2.1 IntegerOutputBase	128
9.10.3 Member Function Documentation	128
9.10.3.1 available()	128
9.10.3.2 find() [1/2]	129
9.10.3.3 find() [2/2]	129
9.10.3.4 findUntil() [1/2]	129
9.10.3.5 findUntil() [2/2]	130
9.10.3.6 peek()	130
9.10.3.7 print() [1/10]	130
9.10.3.8 print() [2/10]	131
9.10.3.9 print() [3/10]	131
9.10.3.10 print() [4/10]	131
9.10.3.11 print() [5/10]	132
9.10.3.12 print() [6/10]	132
9.10.3.13 print() [7/10]	133
9.10.3.14 print() [8/10]	133
9.10.3.15 print() [9/10]	134
9.10.3.16 print() [10/10]	134
9.10.3.17 println() [1/10]	134
9.10.3.18 println() [2/10]	135
9.10.3.19 println() [3/10]	135
9.10.3.20 println() [4/10]	135
9.10.3.21 println() [5/10]	136
9.10.3.22 println() [6/10]	136
9.10.3.23 println() [7/10]	136
9.10.3.24 println() [8/10]	137
9.10.3.25 println() [9/10]	137
9.10.3.26 println() [10/10]	137
9.10.3.27 read()	138
9.10.3.28 readBytes() [1/2]	138
9.10.3.29 readBytes() [2/2]	138
9.10.3.30 readBytesUntil() [1/2]	139
9.10.3.31 readBytesUntil() [2/2]	139
9.10.3.32 readFloat() [1/2]	140
9.10.3.33 readFloat() [2/2]	140
9.10.3.34 readLine()	140
9.10.3.35 readLong() [1/2]	141

9.10.3.36 readLong() [2/2]	141
9.10.3.37 setTimeout()	141
9.10.3.38 start()	142
9.10.3.39 stop()	142
9.10.3.40 write() [1/4]	142
9.10.3.41 write() [2/4]	143
9.10.3.42 write() [3/4]	143
9.10.3.43 write() [4/4]	143
9.11 Serial2 Class Reference	144
9.11.1 Detailed Description	147
9.11.2 Member Enumeration Documentation	147
9.11.2.1 IntegerOutputBase	147
9.11.3 Member Function Documentation	147
9.11.3.1 available()	147
9.11.3.2 find() [1/2]	148
9.11.3.3 find() [2/2]	148
9.11.3.4 findUntil() [1/2]	148
9.11.3.5 findUntil() [2/2]	149
9.11.3.6 peek()	149
9.11.3.7 print() [1/10]	149
9.11.3.8 print() [2/10]	150
9.11.3.9 print() [3/10]	150
9.11.3.10 print() [4/10]	150
9.11.3.11 print() [5/10]	151
9.11.3.12 print() [6/10]	151
9.11.3.13 print() [7/10]	152
9.11.3.14 print() [8/10]	152
9.11.3.15 print() [9/10]	153
9.11.3.16 print() [10/10]	153
9.11.3.17 println() [1/10]	153
9.11.3.18 println() [2/10]	154
9.11.3.19 println() [3/10]	154
9.11.3.20 println() [4/10]	154
9.11.3.21 println() [5/10]	155
9.11.3.22 println() [6/10]	155
9.11.3.23 println() [7/10]	155
9.11.3.24 println() [8/10]	156
9.11.3.25 println() [9/10]	156
9.11.3.26 println() [10/10]	156

9.11.3.27 read()	157
9.11.3.28 readBytes() [1/2]	157
9.11.3.29 readBytes() [2/2]	157
9.11.3.30 readBytesUntil() [1/2]	158
9.11.3.31 readBytesUntil() [2/2]	158
9.11.3.32 readFloat() [1/2]	159
9.11.3.33 readFloat() [2/2]	159
9.11.3.34 readLine()	159
9.11.3.35 readLong() [1/2]	160
9.11.3.36 readLong() [2/2]	160
9.11.3.37 setTimeout()	160
9.11.3.38 start()	161
9.11.3.39 stop()	161
9.11.3.40 write() [1/4]	161
9.11.3.41 write() [2/4]	162
9.11.3.42 write() [3/4]	162
9.11.3.43 write() [4/4]	162
9.12 Serial3 Class Reference	163
9.12.1 Detailed Description	166
9.12.2 Member Enumeration Documentation	166
9.12.2.1 IntegerOutputBase	166
9.12.3 Member Function Documentation	166
9.12.3.1 available()	166
9.12.3.2 find() [1/2]	167
9.12.3.3 find() [2/2]	167
9.12.3.4 findUntil() [1/2]	167
9.12.3.5 findUntil() [2/2]	168
9.12.3.6 peek()	168
9.12.3.7 print() [1/10]	168
9.12.3.8 print() [2/10]	169
9.12.3.9 print() [3/10]	169
9.12.3.10 print() [4/10]	169
9.12.3.11 print() [5/10]	170
9.12.3.12 print() [6/10]	170
9.12.3.13 print() [7/10]	171
9.12.3.14 print() [8/10]	171
9.12.3.15 print() [9/10]	172
9.12.3.16 print() [10/10]	172
9.12.3.17 println() [1/10]	172

9.12.3.18 println() [2/10]	173
9.12.3.19 println() [3/10]	173
9.12.3.20 println() [4/10]	173
9.12.3.21 println() [5/10]	174
9.12.3.22 println() [6/10]	174
9.12.3.23 println() [7/10]	174
9.12.3.24 println() [8/10]	175
9.12.3.25 println() [9/10]	175
9.12.3.26 println() [10/10]	175
9.12.3.27 read()	176
9.12.3.28 readBytes() [1/2]	176
9.12.3.29 readBytes() [2/2]	176
9.12.3.30 readBytesUntil() [1/2]	177
9.12.3.31 readBytesUntil() [2/2]	177
9.12.3.32 readFloat() [1/2]	178
9.12.3.33 readFloat() [2/2]	178
9.12.3.34 readLine()	178
9.12.3.35 readLong() [1/2]	179
9.12.3.36 readLong() [2/2]	179
9.12.3.37 setTimeout()	179
9.12.3.38 start()	180
9.12.3.39 stop()	180
9.12.3.40 write() [1/4]	180
9.12.3.41 write() [2/4]	181
9.12.3.42 write() [3/4]	181
9.12.3.43 write() [4/4]	181
9.13 SPI::SPISettings Class Reference	182
9.13.1 Detailed Description	182
9.13.2 Constructor & Destructor Documentation	182
9.13.2.1 SPISettings()	182
9.13.3 Member Function Documentation	183
9.13.3.1 getSpcr()	183
9.13.3.2 getSpsr()	183
9.14 Writer Class Reference	183
9.14.1 Detailed Description	185
9.14.2 Member Enumeration Documentation	185
9.14.2.1 IntegerOutputBase	185
9.14.3 Member Function Documentation	186
9.14.3.1 print() [1/10]	186

9.14.3.2 print() [2/10]	186
9.14.3.3 print() [3/10]	186
9.14.3.4 print() [4/10]	187
9.14.3.5 print() [5/10]	187
9.14.3.6 print() [6/10]	188
9.14.3.7 print() [7/10]	188
9.14.3.8 print() [8/10]	189
9.14.3.9 print() [9/10]	189
9.14.3.10 print() [10/10]	190
9.14.3.11 println() [1/10]	190
9.14.3.12 println() [2/10]	190
9.14.3.13 println() [3/10]	191
9.14.3.14 println() [4/10]	191
9.14.3.15 println() [5/10]	191
9.14.3.16 println() [6/10]	192
9.14.3.17 println() [7/10]	192
9.14.3.18 println() [8/10]	192
9.14.3.19 println() [9/10]	193
9.14.3.20 println() [10/10]	193
9.14.3.21 write() [1/4]	193
9.14.3.22 write() [2/4]	194
9.14.3.23 write() [3/4]	194
9.14.3.24 write() [4/4]	194
10 File Documentation	195
10.1 abi.h File Reference	195
10.1.1 Detailed Description	195
10.2 abi.h	196
10.3 Analog2Digital.h File Reference	196
10.3.1 Detailed Description	198
10.3.2 Macro Definition Documentation	198
10.3.2.1 readGpioPinAnalog	198
10.3.3 Enumeration Type Documentation	198
10.3.3.1 A2DVoltageReference	198
10.3.4 Function Documentation	199
10.3.4.1 initA2D()	199
10.3.4.2 readA2D()	199
10.3.4.3 readGpioPinAnalogV()	200
10.3.4.4 setA2DVoltageReference()	200

10.3.4.5 setA2DVoltageReference11V()	200
10.3.4.6 setA2DVoltageReference256V()	201
10.3.4.7 setA2DVoltageReferenceAREF()	201
10.3.4.8 setA2DVoltageReferenceAVCC()	201
10.4 Analog2Digital.h	201
10.5 ArduinoMegaPins.h File Reference	203
10.5.1 Detailed Description	204
10.6 ArduinoMegaPins.h	204
10.7 ArduinoPins.h File Reference	206
10.7.1 Detailed Description	206
10.8 ArduinoPins.h	207
10.9 ArduinoUnoPins.h File Reference	207
10.9.1 Detailed Description	208
10.10 ArduinoUnoPins.h	208
10.11 GpioPinMacros.h File Reference	209
10.11.1 Detailed Description	212
10.11.2 Macro Definition Documentation	212
10.11.2.1 getGpioADC	212
10.11.2.2 getGpioCOM	213
10.11.2.3 getGpioDDR	213
10.11.2.4 getGpioMASK	213
10.11.2.5 getGpioOCR	214
10.11.2.6 getGpioPIN	214
10.11.2.7 getGpioPORT	214
10.11.2.8 getGpioTCCR	215
10.11.2.9 GpioPin	215
10.11.2.10 GpioPinAnalog	215
10.11.2.11 GpioPinPwm	216
10.11.2.12 isGpioPinModeInput	216
10.11.2.13 isGpioPinModeOutput	216
10.11.2.14 makeGpioVarFromGpioPin	216
10.11.2.15 makeGpioVarFromGpioPinAnalog	217
10.11.2.16 makeGpioVarFromGpioPinPwm	217
10.11.2.17 readGpioPinDigital	217
10.11.2.18 setGpioPinHigh	218
10.11.2.19 setGpioPinLow	218
10.11.2.20 setGpioPinModeInput	218
10.11.2.21 setGpioPinModeInputPullup	218
10.11.2.22 setGpioPinModeOutput	219

10.11.2.23 writeGpioPinDigital	219
10.11.3 Enumeration Type Documentation	219
10.11.3.1 anonymous enum	219
10.11.4 Function Documentation	219
10.11.4.1 isGpioPinModeInputV()	219
10.11.4.2 isGpioPinModeOutputV()	220
10.11.4.3 readGpioPinDigitalV()	220
10.11.4.4 setGpioPinHighV()	220
10.11.4.5 setGpioPinLowV()	220
10.11.4.6 setGpioPinModeInputPullupV()	221
10.11.4.7 setGpioPinModeInputV()	221
10.11.4.8 setGpioPinModeOutputV()	221
10.11.4.9 writeGpioPinDigitalV()	221
10.12 GpioPinMacros.h	222
10.13 I2cLcd.h File Reference	226
10.13.1 Detailed Description	227
10.14 I2cLcd.h	227
10.15 I2cMaster.h File Reference	230
10.15.1 Detailed Description	232
10.16 I2cMaster.h	233
10.17 I2cSlave.h File Reference	235
10.17.1 Detailed Description	236
10.18 I2cSlave.h	237
10.19 InitSystem.h File Reference	238
10.19.1 Detailed Description	238
10.19.2 Function Documentation	239
10.19.2.1 initSystem()	239
10.20 InitSystem.h	239
10.21 InterruptUtils.h File Reference	239
10.21.1 Detailed Description	241
10.22 InterruptUtils.h	241
10.23 MemUtils.h File Reference	243
10.23.1 Detailed Description	244
10.24 MemUtils.h	244
10.25 new.h File Reference	245
10.25.1 Detailed Description	245
10.26 new.h	246
10.27 Pwm.h File Reference	246
10.27.1 Detailed Description	248

10.27.2 Macro Definition Documentation	249
10.27.2.1 writeGpioPinPwm	249
10.27.3 Function Documentation	249
10.27.3.1 clearTimer0()	249
10.27.3.2 clearTimer1()	250
10.27.3.3 clearTimer2()	250
10.27.3.4 clearTimer3()	250
10.27.3.5 clearTimer4()	250
10.27.3.6 clearTimer5()	251
10.27.3.7 initPwmTimer0()	251
10.27.3.8 initPwmTimer1()	252
10.27.3.9 initPwmTimer2()	252
10.27.3.10 initPwmTimer3()	252
10.27.3.11 initPwmTimer4()	253
10.27.3.12 initPwmTimer5()	253
10.27.3.13 writeGpioPinPwmV()	254
10.28 Pwm.h	254
10.29 Reader.h File Reference	256
10.29.1 Detailed Description	257
10.30 Reader.h	257
10.31 RingBuffer.h File Reference	259
10.31.1 Detailed Description	259
10.32 RingBuffer.h	260
10.33 RingBufferT.h File Reference	261
10.33.1 Detailed Description	261
10.34 RingBufferT.h	261
10.35 SimpleDelays.h File Reference	263
10.35.1 Detailed Description	264
10.35.2 Function Documentation	265
10.35.2.1 delayQuartersOfMicroSeconds()	265
10.35.2.2 delayTenthsOfSeconds()	265
10.35.2.3 delayWholeMilliSeconds()	266
10.36 SimpleDelays.h	266
10.37 SPI.h File Reference	267
10.37.1 Detailed Description	268
10.38 SPI.h	268
10.39 SystemClock.h File Reference	272
10.39.1 Detailed Description	273
10.39.2 Function Documentation	273

10.39.2.1 delay()	273
10.39.2.2 delayMicroseconds()	274
10.39.2.3 delayMilliseconds()	274
10.39.2.4 initSystemClock()	274
10.39.2.5 micros()	274
10.39.2.6 millis()	275
10.40 SystemClock.h	275
10.41 USART0.h File Reference	276
10.41.1 Detailed Description	277
10.41.2 Enumeration Type Documentation	278
10.41.2.1 UsartSerialConfiguration	278
10.42 USART0.h	279
10.43 USART0Minimal.h File Reference	281
10.43.1 Detailed Description	281
10.43.2 Function Documentation	282
10.43.2.1 initUSART0()	282
10.43.2.2 receiveUSART0()	282
10.43.2.3 releaseUSART0()	282
10.43.2.4 transmitUSART0() [1/2]	282
10.43.2.5 transmitUSART0() [2/2]	283
10.44 USART0Minimal.h	283
10.45 USART1.h File Reference	284
10.45.1 Detailed Description	285
10.45.2 Enumeration Type Documentation	286
10.45.2.1 UsartSerialConfiguration	286
10.46 USART1.h	287
10.47 USART1Minimal.h File Reference	289
10.47.1 Detailed Description	289
10.47.2 Function Documentation	290
10.47.2.1 initUSART1()	290
10.47.2.2 receiveUSART1()	290
10.47.2.3 releaseUSART1()	290
10.47.2.4 transmitUSART1() [1/2]	291
10.47.2.5 transmitUSART1() [2/2]	291
10.48 USART1Minimal.h	292
10.49 USART2.h File Reference	292
10.49.1 Detailed Description	294
10.49.2 Enumeration Type Documentation	295
10.49.2.1 UsartSerialConfiguration	295

10.50 USART2.h	296
10.51 USART2Minimal.h File Reference	298
10.51.1 Detailed Description	298
10.51.2 Function Documentation	299
10.51.2.1 initUSART2()	299
10.51.2.2 receiveUSART2()	299
10.51.2.3 releaseUSART2()	299
10.51.2.4 transmitUSART2() [1/2]	300
10.51.2.5 transmitUSART2() [2/2]	300
10.52 USART2Minimal.h	301
10.53 USART3.h File Reference	301
10.53.1 Detailed Description	303
10.53.2 Enumeration Type Documentation	304
10.53.2.1 UsartSerialConfiguration	304
10.54 USART3.h	305
10.55 USART3Minimal.h File Reference	307
10.55.1 Detailed Description	307
10.55.2 Function Documentation	308
10.55.2.1 initUSART3()	308
10.55.2.2 receiveUSART3()	308
10.55.2.3 releaseUSART3()	308
10.55.2.4 transmitUSART3() [1/2]	309
10.55.2.5 transmitUSART3() [2/2]	309
10.56 USART3Minimal.h	310
10.57 Writer.h File Reference	310
10.57.1 Detailed Description	311
10.58 Writer.h	311

Index	315
--------------	------------

Chapter 1

AVRTools: A Library for the AVR ATmega328 and ATmega2560 Microcontrollers

1.1 Overview

This library provides an Arduino-like, simple-to-use interface to the AVR ATmega328 and ATmega2560 microcontrollers without the bloat and slowness of the official Arduino libraries.

AVRTools is an attempt to provide the convenience of the Arduino library interface while embracing the fundamental C/C++ philosophy of "you don't pay for what you don't use" and "assume the programmer knows what he or she is doing."

Like the Arduino libraries, AVRTools allows you to refer to pins on an Arduino via simple names such as `pPin07` for digital pin 7 or `pPinA03` for analog pin 3. However, unlike the Arduino libraries, these names are pure macros so that `setGpioPinHigh(pPin12)` always translates directly into `PORTB |= (1<<PORTB4)` on an Arduino Uno. Similar macros are available for conveniently naming any pin on an ATmega328 or ATmega2560, providing easy and efficient access to all the functionality available on that pin (digital I/O, analog-to-digital conversion, PWM, etc). In combination with these pin name macros, AVRTools provides functions to access the major subsystems and functionality of the ATmega328 and ATmega2560 microcontrollers.

On the other hand, because "you don't pay for what you don't use," when using AVRTools nothing is initialized or configured unless you explicitly do it. If you need analog inputs, then you must explicitly initialize the analog-to-digital subsystem before reading any analog pins. If you need an Arduino-style system clock (for functions like `delay()` or `millis()`), then you must explicitly start a system clock. AVRTools provides functions to do any necessary initialization, but the programmer must explicitly call these function to perform the initialization.

Similarly, because AVRTools "assumes the programmer knows what he or she is doing," it doesn't conduct a lot of checks to ensure you don't do something stupid. For example, when you set the output value of a digital pin using the Arduino library function `digitalWrite()`, it checks if that pin is currently configured for PWM and if it is, it automatically turns off PWM-mode before writing to the pin. The equivalent of `digitalWrite()` in the AVRTools library, `writeGpioPinDigital()` doesn't do that: it assumes that if the programmer previously used the pin in PWM mode that they remembered to turn off PWM mode before using the pin digitally. Assuming the programmer knows what they are doing allows the functions in AVRTools to be much faster than their Arduino library counterparts. For example, a call to the Arduino function `digitalWrite()` takes about 70 cycles; a call to the equivalent AVRTools function `writeGpioPinDigital()` takes 2 cycles (it's actually a macro in AVRTools that the compiler translates to a single, 2-cycle assembler instruction).

1.2 Audience

If you are an Arduino programmer, you may want to try AVRTools if:

- You are comfortable programming the Arduino Uno and Mega directly using the the avr-gcc toolset.
- You are frustrated by the slowness of even simple functions in the official Arduino libraries.
- Your code doesn't fit into the available memory because the official Arduino libraries are so big.

If you are an ATmega328 or ATmega2560 microcontroller programmer, you may want to try AVRTools if:

- You are secretly jealous of how easy and convenient it is to use the Arduino libraries.
- You wish you could bind together DDRs, PORTs, and PINs so you didn't have to write code like:

```
#define MY_PIN_DDR      DDRB
#define MY_PIN_PORT     PORTB
#define MY_PIN_PIN      PINB
#define MY_PIN_NBR      7

/* Put MY_PIN in output mode and set it high */
MY_PIN_DDR |= (1<<MY_PIN_NBR)
MY_PIN_PORT |= (1<<MY_PIN_NBR)
```

- You wish you could use a function-like syntax to switch input/output mode, read a pin, or set a pin high or low but still have the compiler generate simple `in` and `out` type of instructions.

If you fit into either category, then you should read further.

1.3 AVRTools is not...

AVRTools is not a general purpose AVR programming library. I use the Arduino Uno and the Arduino Mega in my projects, and I wrote AVRTools to support these specific needs. There is conditional code throughout the implementation that is tailored to the ATmega328 and ATmega2560 microcontrollers. Additional conditional code could be added to create corresponding implementations for other AVR processors in the AT-family, but I haven't done it. Furthermore, the code is written for (and works with) microcontrollers running at either 8 MHz, 12 MHz, or 16 MHz (the code automatically adapts to these three clock speeds). I have not tested any other clock speeds, and some of the delay functions are specifically coded for 8 MHz, 12 MHz, and 16 MHz and will not work (as written) at other clock speeds.

Finally, the AVRTools interface is designed to meet my needs and coding style. That means the interfaces are designed in ways which may not reflect your usage. A particular example of this is the I2C module, which is designed to support the I2C idioms I use in my projects and is significantly different from the I2C interface offered by the Arduino libraries.

AVRTools is a C++ library. People may say that it is crazy to use C++ to program a microcontroller because C++ adds bloat and overhead, because behind your back the C++ compiler adds lots of code to make unnecessary copies, manage heap objects, handle exceptions, etc. C++, much like C, is a language that rewards programmers who know what they are doing and punishes those who don't. One can use C++ because it is a "better C" and use C++ features without incurring performance penalties or code bloat. For example, AVRTools uses namespaces to compartmentalize functionality into logical units and avoid name clashes; AVRTools also uses classes in a few cases where objects provide the most natural and convenient implementation of a capability (for example, certain advanced output classes such as [USART0](#) or [I2cLcd](#); note that AVRTools also provides a minimalistic USART interface using functions instead of classes, because different needs call for different tools).

1.4 Quick Tour of AVRTools

This section provides an overview of how AVRTools works, starting with the foundational elements and then summarizing the modules that provide interfaces into the major hardware subsystems of the ATmega328 and ATmega2560 microcontrollers.

1.4.1 Foundational Elements and Concepts

The foundation of the AVRTools library consists of a collection of macros that enable you to refer to "pins" on the chips using a single name that can be used to switch input/output mode, read, or write a pin. This single name provides access, as appropriate, to the DDRx, PORTx, PINx registers and also the specific pin number. For pins that support analog-to-digital conversion, the single name also provides access to the analog channel associated with the pin. For pins that support PWM, the single name also provides access to the control and compare registers and bits needed to configure and control the PWM functionality of that pin.

This is all done via preprocessor macros, both for the single pin name mechanism and for the "functions" that make use of that single pin name. This means that access to any pin-related functionality is as fast as possible, designed specifically so that the `avr-gcc` compiler will emit simple 1- or 2-cycle `in`, `out`, `sbi`, `cbi`, `sbic`, or `sbis` instructions for such operations whenever possible. However, the complex internal representation of the macros means that the pin names are strictly constant and can only be passed to the specialized macro-functions designed to manipulate them. Although they may look and feel like simple constants, pin names cannot be assigned to variables, or passed to ordinary C/C++ functions (however, see the [GPIO Pin Variables](#) section in the [Advanced Features](#) section for a way to create and use variables for the GPIO pins). The AVRTools library does include macro-functions to extract any of the components related to a pin name so that users can access and manipulate the individual components as needed.

1.4.2 What you need to know about pin name macros

To access the pin names of the Arduino Uno or Mega, you only need to include the file "ArduinoPins.h". It will automatically detect whether you are compiling for Uno or Mega and it will correspondingly define the macros `pPinNN` (NN = 00 to 13 for Arduino Uno, NN = 00 to 53 for Mega) for digital ports and macros `pPinAnn` (nn = 00 to 07 for Uno, nn = 00 to 15 for Mega) for the analog ports. These correspond directly to the labeled pins on the Arduino boards. You can use these pin names to define your own macros:

```
#define THE_RED_LED      pPin12      // Red LED on Arduino pin 12
#define THE_GRN_LED      pPin11      // Green LED on Arduino pin 11
#define POTENTIOMETER    pPinA03     // Potentiometer on Arduino pin A3
```

While you cannot assign these to pin names to variables or pass them to ordinary functions, AVRTools provides a large collection of macro-functions to operate on the pin names. These include:

- `setGpioPinModeOutput(pin)` Enable the corresponding DDRn bit
- `setGpioPinModeInput(pin)` Clear the corresponding DDRn and PORTn bits
- `setGpioPinModeInputPullup(pin)` Clear the corresponding DDRn bit and set the PORTn bit
- `isGpioPinModeOutput(pin)` Is the corresponding DDRn bit set?
- `isGpioPinModeInput(pin)` Is the corresponding DDRn bit clear?
- `readGpioPinDigital(pin)` Is the corresponding PINn bit is set? (returns zero or non-zero)
- `writeGpioPinDigital(pin, value)` Write a 0 or 1 to the corresponding PORTn bit

- `setGpioPinHigh(pin)` Set the corresponding PORTn bit
- `setGpioPinLow(pin)` Clear the corresponding PORTn bit
- `readGpioPinAnalog(pin)` Read an analog value from the corresponding ADC channel
- `writeGpioPinPwm(pin, value)` Set the corresponding PWM output level for that pin

Most of these macros are automatically defined when you include "ArduinoPins.h", although to define the last two you need to include "Analog2Digital.h" and "Pwm.h" (respectively). These macros allow you to write code such as:

```
// Assuming everything has been initialized properly before this point
```

```
setGpioPinModeOutput( THE_RED_LED );
setGpioPinLow( THE_RED_LED );

setGpioPinModeOutput( THE_GRN_LED );
setGpioPinLow( THE_GRN_LED );

if ( readGpioPinAnalog( POTENTIOMETER ) < 100 )
{
    setGpioPinHigh( THE_RED_LED );
}
else
{
    setGpioPinHigh( THE_GRN_LED );
}
```

If you are working directly with an AVR ATmega328 or ATmega2560, you can define pin macros yourself by including "GpioPinMacros.h" (this file is automatically included for you when you include "ArduinoPins.h" if you are working on Arduinos) and using one of three pin naming macros:

- `GpioPin(letter, number)` An ordinary pin located on bank `letter` and bit `number`; for example the macro `GpioPin(B, 5)` corresponds to pin PB5.
- `GpioPinAnalog(letter, number, channel)` An ADC capable pin on bank `letter` and bit `number` with ADC channel, e.g., `GpioPinAnalog(C, 5, 5)` for ATmega328 pin PC5/ADC5, or `GpioPinAnalog(K, 1, 9)` for ATmega2560 pin PK1/ADC9.
- `GpioPinPwm(letter, number, timer, channel)` A PWM capable pin on bank `letter` and bit `number` with `timer` and `channel` used to select the appropriate OCRn[A/B], TCCRnA registers, and COMn[A/B]1 bits needed to configure the PWM settings, e.g., `GpioPinPwm(B, 2, 1, B)` for ATmega328 pin PB2/OC1B.

So for example, pin 11 on the Arduino Uno, which corresponds to ATmega328 pin B3 which is PWM capable using OC2A, would be defined as follows:

```
#define pPin11 GpioPinPwm( B, 3, 2, A )
```

1.4.3 The core modules

In addition to the macro-based pin naming and access system discussed above, there are seven additional elements that make up the core of AVRTools and provide access to basic functional elements of the ATmega328 and ATmega2560 microcontrollers. Together, these provide an Arduino-like interface to the microcontroller features. Five of the seven modules directly interface to microcontroller capabilities:

- [System initialization module](#)
- [System clock module](#)

- [Analog-to-Digital module](#)
- [PWM module](#)
- [Minimal USART modules](#)

Two of the seven modules supplement the C++ implementation provided by the `avr-gcc` toolset:

- [ABI module](#) (support for the C++ ABI not included in the `avr-gcc` distribution)
- [New module](#) (implementation for `operator new` and `operator delete`)

Brief descriptions of these modules follow.

1.4.3.1 System initialization module

This module provides a single function that puts the microcontroller in a clean, known state. To use it include the header file [InitSystem.h](#) and link against `InitSystem.cpp`. These files provides a single function:

```
void initSystem();
```

The `initSystem()` function clears any bootloader settings, clears all timers, and turns on interrupts. This should be the first function your code calls at start up.

1.4.3.2 System clock module

This module provides a system clock functionality using Timer0 similar to that in the Arduino library. To employ this functionality include the header file [SystemClock.h](#) and link against `SystemClock.cpp`.

Some of key functions provided by this module include:

```
void initSystemClock();  
unsigned long millis();  
void delayMilliseconds( unsigned long ms );
```

Note that unlike the Arduino library, you must explicitly initialize the clock functionality by calling `initSystemClock()`. This module also provides additional functions providing a richer interface to the system clock.

1.4.3.3 Analog-to-Digital module

This module provides access to the analog read capabilities of the ATmega328 and ATmega2560. To employ this functionality include the header file [Analog2Digital.h](#) and link against `Analog2Digital.cpp`. The principle functions provided by this module include:

```
void initA2D();  
void turnOffA2D();  
readGpioPinAnalog( pinName );    /* implemented as a macro */
```

You must initialize the analog-to-digital subsystem by calling `initA2D()` before attempting to read any analog pins.

1.4.3.4 PWM module

This module provides access to the PWM features available on certain ATmega328 and ATmega2560 pins. To employ this functionality include the header file [Pwm.h](#) and link against `Pwm.cpp`. The principle functions provided by this module include (among others):

```
void initPwmTimer1();
void initPwmTimer2();
void clearTimer1();
void clearTimer2();
writeGpioPinPwm( pinName, value ); /* implemented as a macro */
```

Depending on which pins you wish to employ in PWM mode, you should initialize the appropriate timers by calling the appropriate `initPwmTimerN()` function (where N is the appropriate timer number) before writing to the pin in PWM mode. This module also includes additional functions to access the extended PWM capabilities of the ATmega2560. The philosophical difference between the standard Arduino library and AVRTools is evident in this module: none of these function try to deduce which timers need to be turned on for any given pin, because that would require adding extra code and look-up tables. Instead AVRTools assumes the programmer will check the appropriate references to determine which timers correspond to the pins they want to use in PWM mode, and will use that knowledge to initialize the appropriate timers. For convenience, tables of PWM-capable pins and corresponding timers are included in the [Pwm.h](#) documentation.

1.4.3.5 Minimal USART modules

These modules provide basic functionality for reading and writing from the USARTs available on the ATmega328 and ATmega2560. To employ the USART0 functionality, you must include the header file [USART0Minimal.h](#) and link against the file `USART0Minimal.cpp`. The principle functions for accessing the USARTs are:

```
void initUSART0( unsigned long baudRate );
void transmitUSART0( unsigned char data );
void transmitUSART0( const char* data );
unsigned char receiveUSART0();
void releaseUSART0();
```

To make use of the USART0 capability, first call `initUSART0()` to initialize the USART. Then you can use `transmitUSART0()` and `receiveUSART0()` functions to communicate on USART0. When you are done with USART0 and want to use pins 0 and 1 for other purposes, call `releaseUSART0()`. Similar functions are provided to access the other three USARTs available on the ATmega2560; simply include [USARTnMinimal.h](#) and link against the file `USARTnMinimal.cpp`, where $n = 1, 2, \text{ or } 3$. If you want more advanced serial capabilities, checkout the class [Serial0](#) in [USART0.h](#).

1.4.3.6 ABI module

You only need this module if building your code produces link errors regarding missing symbols with strange names like `__cxa_XXX` (where XXX is some unusual string). In that case, simply link your code against `abi.cpp`. These are symbols related to the way the avr-gcc C++ compiler implements abstract virtual functions.

1.4.3.7 New module

This module implements `operator new` and `operator delete`. You only need this if you use `new` and `delete` to manage objects on the heap. Link against `new.cpp` to make use of these operators. AVRTools itself does not make any use of heap objects or operators `new` or `delete`.

1.5 Sample start up code using AVRTools

You can use AVRTools to create an environment that is very similar to the standard Arduino environment. The following sample code illustrates how to do this. The sample code reads a potentiometer and sets both a digital pin and a PWM pin based on the value of the potentiometer.

```
#include "AVRTools/ArduinoPins.h"
#include "AVRTools/InitSystem.h"
#include "AVRTools/SystemClock.h"
#include "AVRTools/Analog2Digital.h"
#include "AVRTools/Pwm.h"

#define pPot          pPinA01
#define pPwmLed       pPin11
#define pLed          pPin04

int main()
{
    initSystem();
    initSystemClock();
    initPwmTimer2();
    initA2D();

    setGpioPinModeOutput( pLed );
    setGpioPinModeOutput( pPwmLed );
    setGpioPinModeInput( pPot );

    while ( 1 )
    {
        int i = readGpioPinAnalog( pPot ) / 4;

        writeGpioPinPwm( pPwmLed, i );

        if ( i > 127 )
        {
            setGpioPinHigh( pLed );
        }
        else
        {
            setGpioPinLow( pLed );
        }

        delayMilliseconds( 100 );
    }
}
```

1.6 Advanced modules

AVRTools also includes modules that provide access to more complex microcontroller capabilities and provide advanced services. These include modules for I2C communication (both master and slave mode), a module for SPI communications, a module for more advanced serial input and output (including conversion of various numerical types and strings), a module for temporarily suppressing selected interrupts, a module for driving an LCD display via I2C, a module for reporting memory utilization, a module for very precise delays, and a module for manipulating GPIO pins as actual variables. Information on these modules can be found in the [Advanced Features](#) sections of the documentation.

1.7 Documentation

Detailed documentation is provided by this PDF document located in the repository, or [online in HTML form](#).

1.8 Questions

If you have questions, please check out the [FAQ](#).

Chapter 2

Advanced Features

The AVRTools library includes a collection more advanced and/or specialized features:

- [Advanced serial \(USART\) module](#)
- [I2C modules](#)
- [I2C-based LCD module](#)
- [Interrupt utilities module](#)
- [SPI module](#)
- [Memory utilities module](#)
- [Simple delays module](#)
- [GPIO pin variables](#)

These features provide functionality that is different from that provided by the Arduino libraries, either in the design of their interface or in the underlying implementation, or both. While the core modules of the AVRTools library are basically independent and can be used individually, these advanced features depend in various ways upon the AVRTools core modules and, sometimes, on each other. These dependencies are highlighted in the corresponding sections.

2.1 Advanced serial (USART) module

The advanced USART module provides two different high-level interfaces to USART0 hardware available on the Arduino Uno (ATmega328) and the Arduino Mega (ATmega2560). These interfaces provide serial input and output that is flexible, buffered, and asynchronous by exploiting the interrupts that are associated with the USART0 hardware. This means the transmit functions return immediately after queuing data in the output buffer for transmission, and the actual transmission happens asynchronously while your code continues to execute. Similarly, data is received asynchronously and placed into the input buffer for your code to read at its convenience.

If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). When receiving data, however, the receive buffer will overwrite itself when it gets full (in a circular, first-in is first-overwritten fashion). You must clear the receive buffer by reading it regularly when

receiving significant amounts of data. The sizes of the transmit and receive buffers can be set at compile time via macro constants.

Two interfaces to USART0 hardware are provided. The first is provided in namespace [USART0](#) and provides a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontroller hardware. However, [USART0](#) functionality is limited to transmitting and receiving byte and character streams. Think of [USART0](#) as a buffered version of the `receiveUSART0()` and `transmitUSART0()` functions provided by the [Minimal USART modules](#).

The second interface is [Serial0](#). [Serial0](#) is the most advanced and capable interface to the USART0 hardware. [Serial0](#) provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously. [Serial0](#) is implemented using [USART0](#), so you may mix the use of [USART0](#) and [Serial0](#) interfaces in your code (although it is not recommended).

To use these the advanced serial capabilities, include the file [USART0.h](#) in your source code and link against the file `USART0.cpp`.

Note

The advanced serial module is incompatible with the minimal interface to USART0. If you link against the file `USART0.cpp` (even if you don't actually use [Serial0](#) or [USART0](#)), do *not* call `initUSART0()` or `releaseUSART0()`; there is no point in any case because the `receiveUSART0()` and `transmitUSART0()` functions won't work. You may, however, use the minimal interface to access [USART1](#), [USART2](#), and/or [USART3](#) while simultaneously using [Serial0](#) and [USART0](#) to access USART0.

Use of the timeout feature in [Serial0](#) and [USART0](#) requires linking against `SystemClock.cpp` and calling the function `initSystemClock()` from your start-up code.

If you are coding for the ATmega2560, you can also use [USART1](#), [USART2](#), and [USART3](#) and/or [Serial1](#), [Serial2](#), and [Serial3](#) to access the USART1, USART2, and USART3 hardware available on the ATmega2560. These work just like [USART0](#) and [Serial0](#). Again, if you link against `USART1.cpp`, `USART2.cpp`, or `USART3.cpp` then you cannot use the corresponding minimal interfaces for USART1, USART2, and USART3, and calling `initUSART1()` or `releaseUSART1()` (or their corresponding equivalents for the other USARTs) will put that USART in an inoperable configuration.

2.2 I2C modules

There are two modules providing different interfaces to the two-wire serial interface (TWI) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560). These modules provide a high-level interface to I2C protocol communications. There are two different modules corresponding to the role within the I2C protocol that your application will use: if your application functions as an I2C "Master", use the [Master](#) module; if your application functions as an I2C "Slave", use the [Slave](#) module.

Note

AVRTools does not support applications that function both as I2C Masters and I2C Slaves. The two I2C modules provided by AVRTools are incompatible and cannot be mixed.

Both modules offer interfaces that are buffered for both input and output, making use of interrupts associated with the TWI hardware. This means the asynchronous transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated TWI hardware. Similarly, data is received asynchronously and placed into the input buffer.

2.2.1 I2C Master module

The [I2C Master module](#) provides I2C-protocol-based interface to the TWI hardware that implements the Master portions of the I2C protocol. The interfaces are buffered for both input and output and operate using interrupts associated with the TWI hardware. This means the asynchronous transmit functions return immediately after queuing data in the output buffer for transmission, and the transmission happens asynchronously, using dedicated TWI hardware. Similarly, data is received asynchronously and placed into the input buffer.

The interface offered by the [I2C Master module](#) is designed around the normal operating modes of the I2C protocol. From a Master device point of view, I2C communications consist of sending a designated device a message to do something, and then either:

- doing nothing because no further action required on the Master's part (e.g., telling the designated device to shutdown).
- transmitting additional data needed by the designated device (e.g., you told the designated device to store some data, next you need to send the data).
- receiving data from the designated device (e.g., telling the designated device to report the current temperature or to read back some data from its memory).

For very simple devices, the receipt of the message itself can suffice to tell it to do something. More commonly, the instruction to the designated device consists of a single byte that passes a "register address" on the device. It is called a register address because it often corresponds directly to a memory register on the device. But it is best to think of it as an instruction code to the designated device (e.g., 0x01 = report the temperature; 0x02 = set the units to either degrees F or degrees C (depending on additional data sent by the Master); 0x03 = report the humidity; etc.).

The interface offered by the [I2C Master module](#) conforms directly to the above I2C paradigm. For convenience, the interface functions come in both synchronous and asynchronous versions. The synchronous versions simply call the asynchronous versions and block internally until the asynchronous operations are complete.

Note

The [I2C Master module](#) is incompatible with the [I2C Slave module](#): you must use and link against only one of the two modules.

2.2.2 I2C Slave module

The [I2C Slave module](#) provides I2C-protocol-based interface to the TWI hardware that implements the Slave portions of the I2C protocol. The interfaces are buffered for both input and output and operate using interrupts associated with the TWI hardware. This means the functions return immediately after queuing data for transmission and the transmission happens asynchronously, using the dedicated TWI hardware. Similarly, data is received asynchronously and placed into a buffer.

The interface offered by the [I2C Slave module](#) is designed around the normal operating modes of the I2C protocol. From a Slave device point of view, I2C communications consist of receiving a message from the Master telling it to do something, and in response:

- Processing the message and taking whatever action is appropriate.
- If that action includes returning data to the Master, queuing the data for transmission.

The interface offered by the [I2C Slave module](#) conforms directly to the above I2C paradigm.

2.3 I2C-based LCD module

The [I2C-based LCD module](#) provides a high-level interface to an LCD offering an I2C interface. The most common variant of this is an HD44780U controlled LCD driven by an MCP23017 that offers an I2C interface (such LCDs are available from Adafruit and SparkFun). This module allows you to write to the LCD much as it if were a serial device and includes the ability to write numbers of various types in various formats. It also lets you detect button presses on the 5-button keypad generally associated with such devices.

Note

The [I2C-based LCD module](#) requires the [I2C Master module](#).

2.4 Interrupt utilities module

It is often necessary to suppress interrupts to avoid conflicts between the main thread of code execution and code that runs in interrupts. While it is easy to suppress all interrupts using the `avr-gcc` built-in `cli()` function, often a more selective approach is desirable. And when interrupts are suppressed, it is also easy to forget to re-enable them.

The [Interrupts](#) module addresses these problems by providing simple utility C++ classes whose constructors disable certain kinds of interrupts and whose corresponding destructors re-enable them. A block of code can suppress interrupts by simply declaring an object of one of these classes; interrupts will be automatically restored when the block of code is exited for any reason. For example, if you want to suppress two of the pin change interrupts in a certain block of code, you would do this:

```
#include "AVRTools/InterruptUtils.h"

//
// ... snip ...
//

void dontLetPinChangeInterruptsHappenHere( uint8_t data )
{
    // Pin change interrupts 1 and 2 conflict with this function, so suppress these
    // two pin change interrupts for the duration of this function
    Interrupts::PinChangeOff interruptsOff( kPinChangeInterrupt1 | kPinChangeInterrupt2 );

    // Here is some code that would conflict with the interrupt routines
    // assigned to pin change interrupt 1 and 2...
    // ... snip ...

    // Pin change interrupts 1 and 2 are automatically restored when this
    // function exits
}
```

One common application for this is when using SPI transmissions in both main thread code and interrupt routines. See the documentation for the [SPI module](#) for an example.

2.5 SPI module

The [SPI](#) module provides a high-level interface to the SPI hardware subsystem present on the AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega) microcontrollers. This module provides functions to initialize the SPI hardware, configure it appropriately for your needs, and transmit (and receive) data. While the SPI hardware supports asynchronous transmission via interrupt functions (analogous to the I2C hardware), AVRTools does not implement asynchronous SPI transmission, instead implementing synchronous transmission that polls the appropriate SPI status register to determine when transmission of a byte has completed. The reason for this is that testing of polling and

interrupt implementations by [Tomaž Šolc](#) has shown that polling implementations are faster than interrupt-based implementations by nearly a factor of 2. This is because SPI can work at half the CPU frequency; at this speed, the CPU can only execute about 16 instructions per byte sent via SPI. When the CPU is calling interrupts so often, the overhead of calling the interrupt function dominates, and is greater than the overhead of a simple polling loop.

The [SPI](#) module only implements SPI operation in master mode. Slave mode SPI operation is not supported at this time. In master mode, you may use any free pin as the Slave Select (SS) for the remote device.

The [SPI](#) module contains functions to enable and disable the SPI hardware. Note that when enabled the SPI hardware takes control of the MOSI, MISO, and CLK pins. It also sets the local SS pin to output mode to prevent inadvertent automatic triggering of slave-mode by the SPI hardware. This happens if a low signal is received on the SS pin. The SS pin can still be used as a general purpose output port, because it doesn't affect SPI operations as long as it remains in output mode.

To use the [SPI](#) module, call the [SPI::enable\(\)](#) function as part of your initialization. Then when you are ready to transmit, configure the hardware appropriately using [SPI::configure\(\)](#), write the receiving device's slave select pin LOW, call [SPI::transmit\(\)](#) (or any one of the related transmit functions) any number of times to transfer data, and finally write the receiving device's SS pin HIGH to indicate that transmission has ended.

One potential complication may occur if you use SPI to transmit data from inside an interrupt routine and also use SPI in the main execution thread. In this situation, you have to make sure that a SPI transmission in the main thread is not interrupted by a SPI transmission from an interrupt routine. You can do this very easily by disabling the appropriate interrupt (or interrupts) during the period of time that an SPI transmission occurs in the main thread. The classes in [Interrupts](#) allow you to selectively disable interrupts.

The following example code illustrates how to do this. Assume that SPI is used by interrupt routines associated with pin change interrupts 0 and 1, and with external interrupt 1. Your main thread code would then look like this:

```
// ... snip ...

#include "AVRTools/SPI.h"
#include "AVRTools/InterruptUtils.h"

// ... snip ...

void initializeEverything()
{
    // ... snip ...

    // Initialize the SPI subsystem
    SPI::enable();
}

// ... snip ...

uint8_t sendData( uint8_t data )
{
    // For illustration, assume SPI is also used by interrupt functions, in particular
    // SPI is used by the interrupt functions that respond to pin change interrupts 0 and 1,
    // and external interrupt 1. To prevent clashes, we suppress these three interrupts
    // for the duration of this function
    Interrupts::PinChangeOff pinChangeOff( kPinChangeInterrupt0 | kPinChangeInterrupt1 );
    Interrupts::ExternalOff externalOff( kExternalInterrupt1 );

    // Configure SPI
    SPI::configure( SPISettings( 4000000, SPI::kLsbFirst, SPI::kSpiMode2 ) );

    // Set the remote slave SS pin low to initiate a transmission
    setGpioPinLow( pConnectedToSlaveSS );

    // Transmit (and receive)
    uint8_t retVal = SPI::transmit( data );

    // Set the remote slave SS pin high to terminate the transmission
    setGpioPinHigh( pConnectedToSlaveSS );

    // Interrupts automatically reset when this function exits
    return retVal;
}
```

2.6 Memory utilities module

The [Memory Utilities module](#) provides functions that report the available memory in SRAM. These help you gauge in real-time whether your application is approaching memory exhaustion or the heap and stack are close to colliding. The primary functions are `freeSRAM()` which returns the number of free bytes remaining in SRAM, and `freeMemoryBetweenHeapAndStack()` which returns the number of free bytes remaining between the top of the heap and top of the stack (recall that these grow towards each other).

2.7 Simple delays module

The [Simple Delays module](#) provides simple delay functions that do not involve timers or interrupts. These functions simply execute a series of nested loops with known and precise timing.

These functions are all implemented directly in assembler to guarantee cycle counts. However, if interrupts are enabled, then the delays will be at least as long as requested, but may actually be longer. Depending on the application, it may be appropriate to disable interrupts prior to calling one of these. The delay functions are:

- `delayQuartersOfMicroSeconds(uint16_t nbrOfQuartersOfMicroSeconds);`
- `delayWholeMilliSeconds(uint8_t nbrOfMilliSeconds);`
- `delayTenthsOfSeconds(uint8_t nbrOfTenthsOfSeconds);`

2.8 GPIO pin variables

There is sometimes a desire to assign GPIO pins to variables. Unfortunately, the pin name macros defined for you when you include [ArduinoPins.h](#) or that you define yourself using `GpioPin()`, `GpioPinAnalog()`, or `GpioPinPwm()` cannot be assigned to variables or used for anything other than passing them to the specialized macro functions designed to handle them. This is normally not a big limitation: the use of GPIO pins is generally encapsulated in functions or classes that function much like software drivers for hardware, hiding the pins from the rest of the application. Treating the pins as macro constants usually works well in such situations. However, there do sometimes arise situations in which it would be convenient to be able to assign GPIO pins to variables and manipulate GPIO pins via those variables.

AVRTools provides a way to convert GPIO pins macros into variables and provides corresponding functions for manipulating those variables. However, this convenience comes at a very significant cost for two reasons.

The first reason is that functions that manipulate AVR I/O registers via variables are inherently slower than those that manipulate them as constants. When using the GPIO pin macros, most operations map directly to `in` and `out` AVR assembler instructions. However, due to the constraints on these instructions, when using variables to pass the pins, the compiler must use slower `ld` and `st` instruction to access the I/O registers (for more on this issue, see the section in the [AVR-GCC FAQ](#)). In addition, when using variables and function calls the bit-shifts needed to generate suitable masks have to be generated at run-time (often using loops) instead of at compile-time.

The second reason is that the variables that store GPIO pins are rather large. On the AVR hardware architecture, manipulating a GPIO pin requires knowing three different I/O registers (`DDRN`, `PORTn`, and `PINn`) and a bit number. Accessing an analog pin requires a corresponding analog-to-digital channel number. Manipulating a PWM pin requires knowing two additional registers (`OCRn[A/B]` and `TCCRnA`) and another bit number (`COMn[A/B]1`). So a general-purpose variable representing a GPIO pin has to store all of these registers, bit numbers, and channel numbers. It is

possible to create smaller GPIO pin variables by encoding information and using look-up tables. The costs are still there, and it is simply a choice of where to pay them. In AVRTools, the choice is to implement "heavy" variables and avoid look-up tables and encoding schemes.

In AVRTools, GPIO pin variables have type `GpioPinVariable`, which is a class defined in `GpioPinMacros.h` (recall that this file is automatically included by `ArduinoPins.h`). There are three macros that you can use to initialize GPIO pin variables of type `GpioPinVariable`. These are: `makeGpioVarFromGpioPin()`, `makeGpioVarFromGpioPinAnalog()`, and `makeGpioVarFromGpioPinPwm()`. They are used like this:

```
GpioPinVariable pinA( makeGpioVarFromGpioPin( pPin10 ) );
GpioPinVariable pinB = makeGpioVarFromGpioPinAnalog( pPinA01 );
GpioPinVariable pinC( makeGpioVarFromGpioPinPwm( pPin03 ) );

GpioPinVariable pinArray[3];
pinArray[0] = pinA;
pinArray[1] = pinB;
pinArray[2] = makeGpioVarFromGpioPin( pPin07 );
```

Which macro you choose depends upon what functionality of the GPIO pin you plan to access: you are free to use `makeGpioVarFromGpioPin()` with an analog pin macro (e.g., `pPinA01`) if you only plan to use the resulting variable digitally. But if you plan to use the analog capabilities of the GPIO pin, you must use `makeGpioVarFromGpioPinAnalog()` to initialize the variable. Similarly for PWM functionality.

Once you've created GPIO pin variables using the above macros, these variables can be assigned and passed to functions as needed. To use these GPIO pin variables, there are special function analogs of the pin manipulation macros. These have the same names as the pin manipulation macros, except with a "V" appended:

Macro Version	Function Version	Purpose
<code>isGpioPinModeOutput(pinMacro)</code>	<code>isGpioPinModeOutputV(const GpioPinVariable& pinVar)</code>	Is the corresponding DDRn bit set?
<code>isGpioPinModeInput(pinMacro)</code>	<code>isGpioPinModeInputV(const GpioPinVariable& pinVar)</code>	Is the corresponding DDRn bit clear?
<code>setGpioPinModeOutput(pinMacro)</code>	<code>setGpioPinModeOutputV(const GpioPinVariable& pinVar)</code>	Enable the pin's corresponding DDRn bit
<code>setGpioPinModeInput(pinMacro)</code>	<code>setGpioPinModeInputV(const GpioPinVariable& pinVar)</code>	Clear the pin's corresponding DDRn bit
<code>setGpioPinModeInputPullup(pinMacro)</code>	<code>setGpioPinModeInputPullupV(const GpioPinVariable& pinVar)</code>	Clear the corresponding DDRn and PORTn bits
<code>readGpioPinDigital(pinMacro)</code>	<code>readGpioPinDigitalV(const GpioPinVariable& pinVar)</code>	Return the value (0 or 1) of the corresponding PINn bit
<code>writeGpioPinDigital(pinMacro, value)</code>	<code>writeGpioPinDigitalV(const GpioPinVariable& pinVar, bool value)</code>	Write a 0 or 1 to the corresponding PORTn bit
<code>setGpioPinHigh(pinMacro)</code>	<code>setGpioPinHighV(const GpioPinVariable& pinVar)</code>	Set the corresponding PORTn bit
<code>setGpioPinLow(pinMacro)</code>	<code>setGpioPinLowV(const GpioPinVariable& pinVar)</code>	Clear the corresponding PORTn bit
<code>readGpioPinAnalog(pinMacro)</code>	<code>readGpioPinAnalogV(const GpioPinVariable& pinVar)</code>	Read an analog value from the corresponding ADC channel
<code>writeGpioPinPwm(pinMacro, value)</code>	<code>writeGpioPinPwmV(const GpioPinVariable& pinVar, uint8_t value)</code>	Set the corresponding PWM output level for that pin

Note

GPIO pin variables can only be passed to the function versions; GPIO pin variables cannot be passed to the macro versions. Similarly, GPIO pin macros cannot be passed to the function versions.

To illustrate how GPIO pin variables can be used, here are two versions of a trivial program, the first using the macros, and the second using variables.

2.8.1 Example using GPIO pin macros

Compiled for an Arduino Uno, the following program is 1,978 bytes.

```
#include "AVRTools/ArduinoPins.h"
#include "AVRTools/InitSystem.h"
#include "AVRTools/SystemClock.h"

#define pRed          pPin10
#define pYellow       pPin07
#define pGreen        pPin04

int main()
{
    initSystem();
    initSystemClock();

    setGpioPinModeOutput( pGreen );
    setGpioPinModeOutput( pYellow );
    setGpioPinModeOutput( pRed );

    setGpioPinHigh( pGreen );
    setGpioPinHigh( pYellow );
    setGpioPinHigh( pRed );

    delayMilliseconds( 2000 );

    setGpioPinLow( pGreen );
    setGpioPinLow( pYellow );
    setGpioPinLow( pRed );

    while ( 1 )
    {
        delayMilliseconds( 1000 );

        setGpioPinLow( pRed );
        setGpioPinHigh( pGreen );

        delayMilliseconds( 1000 );

        setGpioPinLow( pGreen );
        setGpioPinHigh( pYellow );

        delayMilliseconds( 1000 );

        setGpioPinLow( pYellow );
        setGpioPinHigh( pRed );
    }
}
```

2.8.2 Example using GPIO pin variables

Compiled for an Arduino Uno, the following program is 2,456 bytes (478 bytes larger than the macro version) and uses an additional 45 bytes of SRAM compared to the macro version.

```
#include "AVRTools/ArduinoPins.h"
#include "AVRTools/InitSystem.h"
#include "AVRTools/SystemClock.h"

#define pRed          pPin10
#define pYellow       pPin07
#define pGreen        pPin04

int main()
{
    initSystem();
    initSystemClock();

    GpioPinVariable pins[3];
    pins[0] = makeGpioVarFromGpioPin( pRed );
    pins[1] = makeGpioVarFromGpioPin( pYellow );
    pins[2] = makeGpioVarFromGpioPin( pGreen );

    for ( int i = 0; i < 3; i++ )
    {
        setGpioPinModeOutputV( pins[i] );
    }
}
```

```
        setGpioPinHighV( pins[i] );
    }

    delayMilliseconds( 2000 );

    for ( int i = 0; i < 3; i++ )
    {
        setGpioPinLowV( pins[i] );
    }

    int i = 0;
    while ( 1 )
    {
        delayMilliseconds( 1000 );

        setGpioPinLowV( pins[i++] );
        i %= 3;
        setGpioPinHighV( pins[i] );
    }
}
```


Chapter 3

FAQ

3.1 Frequently Asked Questions

- [Can AVRTools be installed as an Arduino IDE Library?](#)
- [Why can't I assign pins like pPin01 to a variable?](#)
- [Why isn't the SPI module asynchronous?](#)
- [Why does the SPI module only implement master mode?](#)
- [Why is there a setGpioPinHigh\(\) macro and a _setGpioPinHigh\(\) macro?](#)
- [_setGpioPinHigh\(\) is defined with 8 arguments, but called with 1 argument—how can that work?](#)
- [Why is there a setGpioPinHigh\(\) macro and a setGpioPinHighV\(\) function?](#)

3.2 Can AVRTools be installed as an Arduino IDE Library?

No, AVRTools is designed to replace the Arduino Library. It is designed for use directly with the `avr-gcc` compiler (the same compiler used by the Arduino IDE).

3.3 Why can't I assign pins like pPin01 to a variable?

Because pin names like `pPin01` are actually complex macros that expand to a comma separated list of other macros. The macro pin names can only be understood and used by the function macros specifically designed to use them. This is explained in greater detail in [What you need to know about pin name macros](#).

If you really need GPIO pin variables, there is a way to do it. See the section on [GPIO pin variables](#). Note in particular that GPIO pin variables come with high costs, both in speed and memory requirements.

3.4 Why isn't the SPI module asynchronous?

The SPI module is implemented synchronously using polling loops because actual testing has shown this to be nearly twice as fast as implementing the functionality asynchronously using interrupts. Tomáš Šolc has done the research and posted the results on his [blog](#). Check it out (and check out his other articles; his blog is pretty interesting).

3.5 Why does the SPI module only implement master mode?

Easy answer: I have never needed anything other than SPI master mode. In every case I use SPI, the AVR microcontroller is the master talking to some external sensor or device that is the slave. While AVR's SPI hardware supports slave mode, I don't think it is common. If you want to use SPI to communicate across two AVR microcontrollers, obviously one of them would have to be in slave mode. But in that situation, I'd probably have them communicate via a serial connection.

If you need a slave mode SPI interface, let me know. It's pretty straightforward to code. I may well get around to doing it one day in any case, just for completeness.

3.6 Why is there a `setGpioPinHigh()` macro and a `_setGpioPinHigh()` macro?

Getting maximum efficiency from the GPIO pin name macros while making them easy to use requires a series of recursive macro expansions. To make this work, it is essential to force rescanning of macro expansions, and using nested macro function calls is a practical way to force macro rescanning. So all of the GPIO pin related macro functions call a helper macro function that has the same name except for a prepended underscore.

The helper macro functions are an internal implementation detail, and that is why they are not formally documented.

3.7 `_setGpioPinHigh()` is defined with 8 arguments, but called with 1 argument—how can that work?

Someone has been reading the header files. It works because of the magic of the C/C++ preprocessor rescanning rules. The rescanning rules are described in 6.10.3.4 of the [ISO Standard for C] (<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>) (the same rules apply to C++). It requires lawyer-like abilities to completely comprehend the full implications of this short paragraph. However, the gist of it is that if you have the following three macros:

```
#define BAR(X,Y) (X+Y)
#define FOO(X) BAR(X)
#define A B,C
```

And if you then call `FOO(A)` in your code, the preprocessor executes the following steps:

- first `FOO(A)` is expanded to `BAR(A)`
- next `BAR(A)` is expanded to `BAR(B,C)`
- then finally `BAR(B,C)` is expanded to `(B+C)`.

This preprocessor rescanning logic is what powers all of the pin macro magic, not just `_setGpioPinHigh()`.

3.8 Why is there a `setGpioPinHigh()` macro and a `setGpioPinHighV()` function?

All of the GPIO pin related "functions" come in two versions. The versions that do not end in a "V" are actually macros and work with the GPIO pin name macros (e.g, `pin01`). The versions that end with a "V" are true functions and work with GPIO variables. See [GPIO pin variables](#).

Chapter 4

Namespace Index

4.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

I2cMaster	This namespace bundles the I2C-protocol-based interface to the TWI hardware. It provides logical cohesion for functions implement the Master portions of the I2C protocol and prevents namespace collisions	33
I2cSlave	This namespace bundles the I2C-protocol-based interface to the TWI hardware. It provides logical cohesion for functions implement the Slave portions of the I2C protocol and prevents namespace collisions	44
Interrupts	This namespace bundles various utility classes designed to suppress selected interrupts using the RAII idiom	48
MemUtils	A namespace providing encapsulation for functions that report the available memory in SRAM . . .	49
SPI	This namespace bundles an interface to the SPI hardware subsystem on the AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega) microcontrollers. It provides logical cohesion for functions implement the Master portion of the SPI protocol and prevents namespace collisions . . .	51
USART0	This namespace bundles a high-level buffered interface to the USART0 hardware. It provides logical cohesion and prevents namespace collisions	56
USART1	This namespace bundles a high-level buffered interface to the USART1 hardware. It provides logical cohesion and prevents namespace collisions	60
USART2	This namespace bundles a high-level buffered interface to the USART2 hardware. It provides logical cohesion and prevents namespace collisions	64
USART3	This namespace bundles a high-level buffered interface to the USART3 hardware. It provides logical cohesion and prevents namespace collisions	68

Chapter 5

Hierarchical Index

5.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Interrupts::AllOff	73
Interrupts::ExternalOff	73
GpioPinVariable	74
Interrupts::PinChangeOff	91
Reader	92
Serial0	106
Serial1	125
Serial2	144
Serial3	163
RingBuffer	99
RingBufferT< T, N, SIZE >	102
SPI::SPISettings	182
Writer	183
I2cLcd	76
Serial0	106
Serial1	125
Serial2	144
Serial3	163

Chapter 6

Class Index

6.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Interrupts::AllOff	This class defines an object that disables all interrupts during its lifetime. Interrupt state is restored by the object's destructor when the object goes out of scope	73
Interrupts::ExternalOff	This class defines an object that disables selected external interrupts during its lifetime. The selected external interrupts are restored by the object's destructor when it goes out of scope	73
GpioPinVariable	This class defines a type that can encode a GPIO pin as a variable. Read the section on [GPIO Pin Variables] (GPIO pin variables) to understand how to use this class	74
I2cLcd	This class provides a high-level interface via I2C to an LCD such as those offered by AdaFruit and SparkFun. Specifically, it communicates via I2C with an MCP23017 that drives an HD44780U controlling an LCD. It also lets you detect button presses on the 5-button keypad generally associated with such devices	76
Interrupts::PinChangeOff	This class defines an object that disables selected pin change interrupts during its lifetime. The selected pin change interrupts are restored by the object's destructor when it goes out of scope . . .	91
Reader	This is an abstract class defining a generic interface to read numbers and strings from a sequential stream of bytes (such as a serial device)	92
RingBuffer	This class provides an efficient ring buffer implementation for storing bytes. Ring buffers are particularly useful for memory constrained microcontrollers such as the ATmega328 and ATmega2650. For maximum efficiency, this class is focused on the storage of bytes, providing a single code base that is shared by all instances of this class	99
RingBufferT< T, N, SIZE >	Template-based ring buffer class that can store different kinds of objects in buffers of whatever size is needed	102
Serial0	Provides a high-end interface to serial communications using USART0	106
Serial1	Provides a high-end interface to serial communications using USART1	125

Serial2	Provides a high-end interface to serial communications using USART2	144
Serial3	Provides a high-end interface to serial communications using USART3	163
SPI::SPISettings	A class that binds settings for configuring SPI transmissions	182
Writer	This is an abstract class defining a generic interface to write numbers and strings to a sequential stream of bytes (such as a serial output device)	183

Chapter 7

File Index

7.1 File List

Here is a list of all documented files with brief descriptions:

abi.h	This file provides certain functions needed to complete the avr-gcc C++ ABI. You never need to include this file, and you only need to link against abi.cpp if you encounter certain link errors	195
Analog2Digital.h	This file provides functions that access the analog-to-digital conversion capability of the ATmega328 and ATmega2560 microcontrollers	196
ArduinoMegaPins.h	This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file ArduinoPins.h , which in turn includes this file (when compiling for Arduino Uno targets)	203
ArduinoPins.h	This file is the primary one that users should include to access and use the pin name macros	206
ArduinoUnoPins.h	This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file ArduinoPins.h , which in turn includes this file (when compiling for Arduino Uno targets)	207
GpioPinMacros.h	This file contains the primary macros for naming and manipulating GPIO pin names	209
I2cLcd.h	This file defines a class that provides a high-level interface to an LCD offering an I2C interface. The most common variant of this is HD44780U controlled LCD driven via an MCP23017 that offers an I2C interface (such LCDs are available from Adafruit and SparkFun). To use this class you must also use and properly initialize the I2C Master package from I2cMaster.h	226
I2cMaster.h	This file provides functions that interface to the TWI (two-wire serial interface) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. Include this file if you want your application will operate in Master mode as defined in the I2C protocol	230
I2cSlave.h	This file provides functions that interface to the TWI (two-wire serial interface) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. Include this file if you want your application will operate in Slave mode as defined in the I2C protocol	235

InitSystem.h	Include this file to use the functions that initialize the microcontroller to a known, basic state	238
InterruptUtils.h	This file provides utilities for temporarily disabling (suppressing) interrupts of various kinds in a block of code. It uses the C++ RAII paradigm to ensure interrupt state is restored automatically when the block of code is exited. While all interrupts can be suppressed, tools are provided that allow more selective control of which interrupts are suppressed	239
MemUtils.h	This file provides functions that provide information on the available memory in SRAM	243
new.h	This file provides <code>operator new</code> and <code>operator delete</code> . You only need this file if you use <code>new</code> and <code>delete</code> to manage objects on the heap	245
Pwm.h	This file provides functions that access the PWM capability of the ATmega328 and ATmega2560 microcontrollers	246
Reader.h	This file provides a generic interface to incoming data streams of any kind. It is designed around how serial streams are generally used, but can be used with any system that provides a sequential input of bytes that can be interpreted as strings and/or numbers	256
RingBuffer.h	This file provides an efficient ring buffer implementation for storing bytes	259
RingBufferT.h	This file provides a very flexible, template-based ring buffer implementation	261
SimpleDelays.h	This file provides simple delay functions that do not involve timers or interrupts. These functions simply execute a series of nested loops with known and precise timing	263
SPI.h	This file provides an interface to SPI subsystem available on the AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega) microcontrollers	267
SystemClock.h	Include this file to use the functions that instantiate and access a system clock that counts elapsed milliseconds	272
USART0.h	This file provides functions that offer high-level interfaces to USART0 hardware, which is available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560)	276
USART0Minimal.h	This file provides functions that provide a minimalist interface to USART0 available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560)	281
USART1.h	This file provides functions that offer high-level interfaces to USART1 hardware, which is available on Arduino Mega (ATmega2560)	284
USART1Minimal.h	This file provides functions that provide a minimalist interface to USART1 available on the Arduino Mega (ATmega2560)	289
USART2.h	This file provides functions that offer high-level interfaces to USART2 hardware, which is available on Arduino Mega (ATmega2560)	292
USART2Minimal.h	This file provides functions that provide a minimalist interface to USART2 available on the Arduino Mega (ATmega2560)	298
USART3.h	This file provides functions that offer high-level interfaces to USART3 hardware, which is available on Arduino Mega (ATmega2560)	301

[USART3Minimal.h](#)

This file provides functions that provide a minimalist interface to [USART3](#) available on the Arduino Mega (ATmega2560) 307

[Writer.h](#)

This file provides a generic interface to outgoing data streams of any kind. It is designed around how serial streams are generally used, but can be used with any system that requires converting strings and/or numbers into a sequential output of bytes 310

Chapter 8

Namespace Documentation

8.1 I2cMaster Namespace Reference

This namespace bundles the I2C-protocol-based interface to the TWI hardware. It provides logical cohesion for functions implement the Master portions of the I2C protocol and prevents namespace collisions.

Enumerations

- enum [I2cBusSpeed](#) { [kI2cBusSlow](#) , [kI2cBusFast](#) }
This enum lists I2C bus speed configurations.
- enum [I2cStatusCodes](#) { [kI2cCompletedOk](#) = 0x00 , [kI2cError](#) = 0x01 , [kI2cNotStarted](#) = 0x02 , [kI2cInProgress](#) = 0x04 }
This enum lists I2C status codes reported by the various transmit functions.
- enum [I2cSendErrorCodes](#) {
[kI2cNoError](#) = 0 , [kI2cErrTxBufferFull](#) = 1 , [kI2cErrMsgTooLong](#) = 2 , [kI2cErrNullStatusPtr](#) = 3 ,
[kI2cErrWriteWithoutData](#) = 4 , [kI2cErrReadWithoutStorage](#) = 5 }
This enum lists I2C errors codes that may occur when you try to write a message.
- enum [I2cPullups](#) { [kPullupsOff](#) , [kPullupsOn](#) }
This enum lists the options for controlling the built-in pullups in the TWI hardware.

Functions

- void [start](#) (uint8_t speed=[kI2cBusFast](#))
Configures the TWI hardware for I2C communications in Master mode. You must call this function before conducting any I2C communications using the functions in this module.
- void [stop](#) ()
Terminates the I2C communications using the TWI hardware, and disables the TWI interrupts.
- void [pullups](#) (uint8_t set=[kPullupsOn](#))
Sets the state of the internal pullups that are part of the TWI hardware.
- bool [busy](#) ()
Reports whether the TWI hardware is busy communicating (either transmitting or receiving).

- `uint8_t writeAsync (uint8_t address, uint8_t registerAddress, volatile uint8_t *status)`
Transmit a single register address (a one-byte message) asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- `uint8_t writeAsync (uint8_t address, uint8_t registerAddress, uint8_t data, volatile uint8_t *status)`
Transmit a single register address and corresponding single byte of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- `uint8_t writeAsync (uint8_t address, uint8_t registerAddress, const char *data, volatile uint8_t *status)`
Transmit a single register address and corresponding null-terminated string of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- `uint8_t writeAsync (uint8_t address, uint8_t registerAddress, uint8_t *data, uint8_t numberBytes, volatile uint8_t *status)`
Transmit a single register address and corresponding buffer of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- `uint8_t readAsync (uint8_t address, uint8_t numberBytes, volatile uint8_t *destination, volatile uint8_t *bytesRead, volatile uint8_t *status)`
Request to read data from a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports `kl2cCompletedOk`, the requested data can be read from the receive buffer.
- `uint8_t readAsync (uint8_t address, uint8_t registerAddress, uint8_t numberBytes, volatile uint8_t *destination, volatile uint8_t *bytesRead, volatile uint8_t *status)`
Request to read data from a specific register on a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports `kl2cCompletedOk`, the requested data can be read from the receive buffer.
- `int writeSync (uint8_t address, uint8_t registerAddress)`
Transmit a single register address (a one-byte message) synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- `int writeSync (uint8_t address, uint8_t registerAddress, uint8_t data)`
Transmit a single register address and corresponding single byte of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- `int writeSync (uint8_t address, uint8_t registerAddress, const char *data)`
Transmit a single register address and corresponding null-terminated string of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- `int writeSync (uint8_t address, uint8_t registerAddress, uint8_t *data, uint8_t numberBytes)`
Transmit a single register address and corresponding buffer of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- `int readSync (uint8_t address, uint8_t numberBytes, uint8_t *destination)`
Request to read data from a device and receive that data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- `int readSync (uint8_t address, uint8_t registerAddress, uint8_t numberBytes, uint8_t *destination)`
Request to read data from a specific register on a device and receive that data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

8.1.1 Detailed Description

This namespace bundles the I2C-protocol-based interface to the TWI hardware. It provides logical cohesion for functions implement the Master portions of the I2C protocol and prevents namespace collisions.

These interfaces are buffered for both input and output and operate using interrupts associated with the TWI hardware. This means the asynchronous transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated TWI hardware. Similarly, data is received asynchronously and placed into the input buffer.

These functions are designed around the normal operating modes of the I2C protocol. From a Master device point of view, I2C communications consist of sending a designated device a message to do something, and then either:

- doing nothing because no further action required on the Master's part (e.g., telling the designated device to shutdown)
- transmitting additional data needed by the designated device (e.g., telling the designated device to store some data)
- receiving data from the designated device (e.g., telling the designated device to report the current temperature or to read back some data from its memory)

For very simple devices, the receipt of the message itself can suffice to tell it to do something. More commonly, the instruction to the designated device consists of a single byte that passes a "register address" on the device. It is call a register address because it often corresponds directly to a memory register on the device. But it is best to think of it as an instruction code to the designated device (e.g., 0x01 = report the temperature; 0x02 = set the units to either F or C (depending on additional data sent by the Master); 0x03 = report the humidity; etc.)

The functions defined by this module conform directly to the above I2C paradigm. The functions come in both synchronous and asynchronous versions. The synchronous versions simply call the asynchronous versions and block internally until the asynchronous operations are complete.

Note also that even "read" operations always begin (from the Master's point of view) with a "send" to the designated device the Master wants to read data from. For this reason all operations (both read and write) utilize the transmit buffer.

8.1.2 Enumeration Type Documentation

8.1.2.1 I2cBusSpeed

```
enum I2cMaster::I2cBusSpeed
```

This enum lists I2C bus speed configurations.

Enumerator

kI2cBusSlow	I2C slow (standard) mode: 100 KHz.
kI2cBusFast	I2C fast mode: 400 KHz.

8.1.2.2 I2cPullups

```
enum I2cMaster::I2cPullups
```

This enum lists the options for controlling the built-in pullups in the TWI hardware.

Enumerator

kPullupsOff	Disable the built-in TWI hardware pullups.
kPullupsOn	Enable the built-in TWI hardware pullups.

8.1.2.3 I2cSendErrorCodes

```
enum I2cMaster::I2cSendErrorCodes
```

This enum lists I2C errors codes that may occur when you try to write a message.

Enumerator

kl2cNoError	No error.
kl2cErrTxBufferFull	The transmit buffer is full (try again later)
kl2cErrMsgTooLong	The message is too long for the transmit buffer.
kl2cErrNullStatusPtr	The pointer to the status variable is null (need to provide a valid pointer)
kl2cErrWriteWithoutData	No data provided to send.
kl2cErrReadWithoutStorage	Performing a write+read, but no buffer provided to store the "read" data.

8.1.2.4 I2cStatusCodes

```
enum I2cMaster::I2cStatusCodes
```

This enum lists I2C status codes reported by the various transmit functions.

Enumerator

kl2cCompletedOk	I2C communications completed on this message with no error.
kl2cError	I2C communications had an error on this message.
kl2cNotStarted	I2C communications not started on this message.
kl2cInProgress	I2C communications on this message still in progress.

8.1.3 Function Documentation

8.1.3.1 busy()

```
bool I2cMaster::busy ( )
```

Reports whether the TWI hardware is busy communicating (either transmitting or receiving).

Returns

true if the TWI hardware is busy communicating; false if the TWI hardware is idle.

8.1.3.2 pullups()

```
void I2cMaster::pullups (
    uint8_t set = kPullupsOn )
```

Sets the state of the internal pullups that are part of the TWI hardware.

[start\(\)](#) automatically enables the internal pullups. You only need to call this function if you want to turn them off, or if you want to alter their state.

- `set` the desired state of the built-in internal pullup. Defaults to enable (`kPullupsOn`).

8.1.3.3 readAsync() [1/2]

```
uint8_t I2cMaster::readAsync (
    uint8_t address,
    uint8_t numberBytes,
    volatile uint8_t * destination,
    volatile uint8_t * bytesRead,
    volatile uint8_t * status )
```

Request to read data from a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports `kI2cCompletedOk`, the requested data can be read from the receive buffer.

If the transmit buffer is full, this function will block until room is available in the buffer.

- `address` the I2C address of the destination device you want to read from.
- `numberBytes` the number of bytes you expect to read.
- `destination` a pointer to a buffer in which the received data will be stored; the buffer should be at least `numberBytes` large.
- `bytesRead` a pointer to a byte-sized counter in which the TWI hardware will asynchronously keep track of how many bytes have been received.
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be updated asynchronously after the function returns by the TWI hardware) values correspond to `I2cStatusCodes`.

Returns

error codes corresponding to `I2cSendErrorCodes` (0 means no error)

8.1.3.4 readAsync() [2/2]

```
uint8_t I2cMaster::readAsync (
    uint8_t address,
    uint8_t registerAddress,
    uint8_t numberBytes,
    volatile uint8_t * destination,
    volatile uint8_t * bytesRead,
    volatile uint8_t * status )
```

Request to read data from a specific register on a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports `kl2cCompletedOk`, the requested data can be read from the receive buffer.

If the transmit buffer is full, this function will block until room is available in the buffer.

- `address` the I2C address of the destination device you want to read from.
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it what you want to read (e.g., temperature or the starting address of a block of memory).
- `numberBytes` the number of bytes you expect to read.
- `destination` a pointer to a buffer in which the received data will be stored; the buffer should be at least `numberBytes` large.
- `bytesRead` a pointer to a byte-sized counter in which the TWI hardware will asynchronously keep track of how many bytes have been received.
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be updated asynchronously after the function returns by the TWI hardware); values correspond to `I2cStatusCodes`.

Returns

error codes corresponding to `I2cSendErrorCodes` (0 means no error)

8.1.3.5 readSync() [1/2]

```
int I2cMaster::readSync (
    uint8_t address,
    uint8_t numberBytes,
    uint8_t * destination )
```

Request to read data from a device and receive that data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

- `address` the I2C address of the destination device you want to read from.
- `numberBytes` the number of bytes you expect to read.
- `destination` a pointer to a buffer in which the received data will be stored; the buffer should be at least `numberBytes` large.

Returns

an error code which if positive corresponds to `I2cSendErrorCodes`, or if negative the absolute value corresponds to `I2cStatusCodes` (0 means no error).

8.1.3.6 readSync() [2/2]

```
int I2cMaster::readSync (
    uint8_t address,
    uint8_t registerAddress,
    uint8_t numberBytes,
    uint8_t * destination )
```

Request to read data from a specific register on a device and receive that data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

- `address` the I2C address of the destination device you want to read from.
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it what you want to read (e.g., temperature or the starting address of a block of memory).
- `numberBytes` the number of bytes you expect to read.
- `destination` a pointer to a buffer in which the received data will be stored; the buffer should be at least `numberBytes` large.

Returns

an error code which if positive corresponds to `I2cSendErrorCodes`, or if negative the absolute value corresponds to `I2cStatusCodes` (0 means no error).

8.1.3.7 start()

```
void I2cMaster::start (
    uint8_t speed = kI2cBusFast )
```

Configures the TWI hardware for I2C communications in Master mode. You must call this function before conducting any I2C communications using the functions in this module.

This function enables the TWI related interrupts and enables the built-in hardware pullups.

- `speed` the speed mode for the I2C protocol. The options are slow (100 KHz) or fast (400 KHz); the default is fast (`kI2cBusFast`).

8.1.3.8 stop()

```
void I2cMaster::stop ( )
```

Terminates the I2C communications using the TWI hardware, and disables the TWI interrupts.

After calling this function, you need to call `start()` again if you want to resume I2C communications.

8.1.3.9 writeAsync() [1/4]

```
uint8_t I2cMaster::writeAsync (
    uint8_t address,
    uint8_t registerAddress,
    const char * data,
    volatile uint8_t * status )
```

Transmit a single register address and corresponding null-terminated string of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).

If the transmit buffer is full, this function will block until room is available in the buffer.

- `address` the I2C address of the destination device for this message
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., an address in a memory device).
- `data` a null-terminated string of data serving as a parameter to the register address (e.g., a string to store sequentially starting at the `registerAddress`).
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be updated asynchronously after the function returns by the TWI hardware); values correspond to `I2cStatusCodes`.

Returns

error codes corresponding to `I2cSendErrorCodes` (0 means no error)

8.1.3.10 writeAsync() [2/4]

```
uint8_t I2cMaster::writeAsync (
    uint8_t address,
    uint8_t registerAddress,
    uint8_t * data,
    uint8_t numberBytes,
    volatile uint8_t * status )
```

Transmit a single register address and corresponding buffer of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).

If the transmit buffer is full, this function will block until room is available in the buffer.

- `address` the I2C address of the destination device for this message
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., an address in a memory device).

- `data` a buffer of data serving as a parameter to the register address (e.g., the data to store sequentially starting at the `registerAddress`).
- `numberBytes` the number of bytes from the buffer to transmit.
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be updated asynchronously after the function returns by the TWI hardware); values correspond to `I2cStatusCodes`.

Returns

error codes corresponding to `I2cSendErrorCodes` (0 means no error)

8.1.3.11 `writeAsync()` [3/4]

```
uint8_t I2cMaster::writeAsync (
    uint8_t address,
    uint8_t registerAddress,
    uint8_t data,
    volatile uint8_t * status )
```

Transmit a single register address and corresponding single byte of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).

If the transmit buffer is full, this function will block until room is available in the buffer.

- `address` the I2C address of the destination device for this message
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., set the volume level).
- `data` a single byte of data serving as a parameter to the register address (e.g., the volume level to set).
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be updated asynchronously after the function returns by the TWI hardware); values correspond to `I2cStatusCodes`.

Returns

error codes corresponding to `I2cSendErrorCodes` (0 means no error)

8.1.3.12 writeAsync() [4/4]

```
uint8_t I2cMaster::writeAsync (
    uint8_t address,
    uint8_t registerAddress,
    volatile uint8_t * status )
```

Transmit a single register address (a one-byte message) asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).

If the transmit buffer is full, this function will block until room is available in the buffer.

- `address` the I2C address of the destination device for this message
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., turn off or on).
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be updated asynchronously after the function returns by the TWI hardware); values correspond to `I2cStatusCodes`.

Returns

error codes corresponding to `I2cSendErrorCodes` (0 means no error)

8.1.3.13 writeSync() [1/4]

```
int I2cMaster::writeSync (
    uint8_t address,
    uint8_t registerAddress )
```

Transmit a single register address (a one-byte message) synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

- `address` the I2C address of the destination device for this message.
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., turn off or on).

Returns

an error code which if positive corresponds to `I2cSendErrorCodes`, or if negative the absolute value corresponds to `I2cStatusCodes` (0 means no error).

8.1.3.14 writeSync() [2/4]

```
int I2cMaster::writeSync (
    uint8_t address,
    uint8_t registerAddress,
    const char * data )
```

Transmit a single register address and corresponding null-terminated string of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

- `address` the I2C address of the destination device for this message
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., an address in a memory device).
- `data` a null-terminated string of data serving as a parameter to the register address (e.g., a string to store sequentially starting at the `registerAddress`).

Returns

an error code which if positive corresponds to `I2cSendErrorCodes`, or if negative the absolute value corresponds to `I2cStatusCodes` (0 means no error).

8.1.3.15 writeSync() [3/4]

```
int I2cMaster::writeSync (
    uint8_t address,
    uint8_t registerAddress,
    uint8_t * data,
    uint8_t numberBytes )
```

Transmit a single register address and corresponding buffer of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

- `address` the I2C address of the destination device for this message
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., an address in a memory device).
- `data` a buffer of data serving as a parameter to the register address (e.g., the data to store sequentially starting at the `registerAddress`).
- `numberBytes` the number of bytes from the buffer to transmit.

Returns

an error code which if positive corresponds to `I2cSendErrorCodes`, or if negative the absolute value corresponds to `I2cStatusCodes` (0 means no error).

8.1.3.16 writeSync() [4/4]

```
int I2cMaster::writeSync (
    uint8_t address,
    uint8_t registerAddress,
    uint8_t data )
```

Transmit a single register address and corresponding single byte of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

- `address` the I2C address of the destination device for this message.
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., set the volume level).
- `data` a single byte of data serving as a parameter to the register address (e.g., the volume level to set).

Returns

an error code which if positive corresponds to `I2cSendErrorCodes`, or if negative the absolute value corresponds to `I2cStatusCodes` (0 means no error).

8.2 I2cSlave Namespace Reference

This namespace bundles the I2C-protocol-based interface to the TWI hardware. It provides logical cohesion for functions implement the Slave portions of the I2C protocol and prevents namespace collisions.

Enumerations

- enum `I2cBusSpeed` { `kl2cBusSlow` , `kl2cBusFast` }
This enum lists I2C bus speed configurations.
- enum `I2cStatusCodes` {
 `kl2cCompletedOk` = 0x00 , `kl2cError` = 0x01 , `kl2cTxPartial` = 0x02 , `kl2cRxOverflow` = 0x04 ,
 `kl2cInProgress` = 0x06 }
This enum lists I2C status codes reported by the various transmit functions.
- enum `I2cPullups` { `kPullupsOff` , `kPullupsOn` }
This enum lists the options for controlling the built-in pullups in the TWI hardware.

Functions

- uint8_t `processI2cMessage` (uint8_t *buffer, uint8_t len)
This function must be defined by the user. It is called by the TWI interrupt function installed as part of `I2cSlave.cpp` whenever it receives a message from the Master. The user should implement this function to respond to the data in the buffer, taking actions and as appropriate returning data to the buffer (for asynchronous transmission to the Master).
- void `start` (uint8_t ownAddress, uint8_t speed=`kl2cBusFast`, bool answerGeneralCall=false)
Configures the TWI hardware for I2C communications in Slave mode. You must call this function before conducting any I2C communications using the functions in this module.
- void `stop` ()
Terminates the I2C communications using the TWI hardware, and disables the TWI interrupts.
- void `pullups` (uint8_t set=`kPullupsOn`)
Sets the state of the internal pullups that are part of the TWI hardware.
- bool `busy` ()
Reports whether the TWI hardware is busy communicating (either transmitting or receiving).

8.2.1 Detailed Description

This namespace bundles the I2C-protocol-based interface to the TWI hardware. It provides logical cohesion for functions implement the Slave portions of the I2C protocol and prevents namespace collisions.

These interfaces are buffered for both input and output and operate using interrupts associated with the TWI hardware. This means the functions return immediately after queuing data for transmission and the transmission happens asynchronously, using the dedicated TWI hardware.

These functions are designed around the normal operating modes of the I2C protocol. From a Slave device point of view, I2C communications consist of receiving a message from the Master telling it to do something, and in response:

- Processing the message and taking whatever action is appropriate.
- If that action includes returning data to the Master, queuing that data for transmission.

The functions defined by this module conform directly to the above I2C paradigm. The key function is [processI2cMessage\(\)](#) and must be defined by the user. This function is called whenever the Slave receives a message and is also used to pass back any data that should be transmitted back to the Master.

8.2.2 Enumeration Type Documentation

8.2.2.1 I2cBusSpeed

```
enum I2cSlave::I2cBusSpeed
```

This enum lists I2C bus speed configurations.

Enumerator

kI2cBusSlow	I2C slow (standard) mode: 100 KHz.
kI2cBusFast	I2C fast mode: 400 KHz.

8.2.2.2 I2cPullups

```
enum I2cSlave::I2cPullups
```

This enum lists the options for controlling the built-in pullups in the TWI hardware.

Enumerator

kPullupsOff	Disable the built-in TWI hardware pullups.
kPullupsOn	Enable the built-in TWI hardware pullups.

8.2.2.3 I2cStatusCodes

```
enum I2cSlave::I2cStatusCodes
```

This enum lists I2C status codes reported by the various transmit functions.

Enumerator

kI2cCompletedOk	I2C communications completed with no error.
kI2cError	I2C communications encountered an error.
kI2cTxPartial	I2C Master terminated transmission before all data were sent.
kI2cRxOverflow	Received a message larger than can be held in the receive buffer.
kI2cInProgress	I2C communications on this message still in progress.

8.2.3 Function Documentation

8.2.3.1 busy()

```
bool I2cSlave::busy ( )
```

Reports whether the TWI hardware is busy communicating (either transmitting or receiving).

Returns

true if the TWI hardware is busy communicating; false if the TWI hardware is idle.

8.2.3.2 processI2cMessage()

```
uint8_t I2cSlave::processI2cMessage (
    uint8_t * buffer,
    uint8_t len )
```

This function must be defined by the user. It is called by the TWI interrupt function installed as part of I2cSlave.cpp whenever it receives a message from the Master. The user should implement this function to respond to the data in the buffer, taking actions and as appropriate returning data to the buffer (for asynchronous transmission to the Master).

The user should implement this function to do the following:

- review the incoming data from the Master
- take appropriate actions in response to that data
- if data must be returned to the Master, write the data into the buffer and return the number of bytes you placed in the buffer
- if no data must be returned to the Master, return 0

Note

This function is called at interrupt time, so the implementation must be kept short. If any significant work must be done as a result of the message received from the Master, this function should simply set a flag that can be detected by the main execution thread and have it do the heavy lifting.

- `buffer` is both an input and output parameter. On entrance to the function, it contains the message received from the Master; on return from the function should contain data (if any) that should be sent back to the Master.
- `len` is only an input parameter. It is the number of received bytes in the input buffer.

Returns

the number of bytes placed in the `buffer` to be sent back to the Master; 0 if no data is to be returned to the Master.

8.2.3.3 pullups()

```
void I2cSlave::pullups (
    uint8_t set = kPullupsOn )
```

Sets the state of the internal pullups that are part of the TWI hardware.

[start\(\)](#) automatically enables the internal pullups. You only need to call this function if you want to turn them off, or if you want to alter their state.

- `set` the desired state of the built-in internal pullup. Defaults to enable (`kPullupsOn`).

8.2.3.4 start()

```
void I2cSlave::start (
    uint8_t ownAddress,
    uint8_t speed = kI2cBusFast,
    bool answerGeneralCall = false )
```

Configures the TWI hardware for I2C communications in Slave mode. You must call this function before conducting any I2C communications using the functions in this module.

This function enables the TWI related interrupts and enables the built-in hardware pullups.

- `ownAddress` is the I2C address for this slave.
- `speed` the speed mode for the I2C protocol. The options are slow (100 KHz) or fast (400 KHz); the default is fast (`kI2cBusFast`).
- `answerGeneralCall` pass true for the Slave to answer I2C general calls; false for the Slave to ignore I2C general calls and only answer calls to his specific address. The defaults is to not answer general calls. and defaults to not answering I2C general calls.

8.2.3.5 stop()

```
void I2cSlave::stop ( )
```

Terminates the I2C communications using the TWI hardware, and disables the TWI interrupts.

After calling this function, you need to call [start\(\)](#) again if you want to resume I2C communications.

8.3 Interrupts Namespace Reference

This namespace bundles various utility classes designed to suppress selected interrupts using the RAII idiom.

Classes

- class [AllOff](#)
This class defines an object that disables all interrupts during its lifetime. Interrupt state is restored by the object's destructor when the object goes out of scope.
- class [ExternalOff](#)
This class defines an object that disables selected external interrupts during its lifetime. The selected external interrupts are restored by the object's destructor when it goes out of scope.
- class [PinChangeOff](#)
This class defines an object that disables selected pin change interrupts during its lifetime. The selected pin change interrupts are restored by the object's destructor when it goes out of scope.

Enumerations

- enum [ExternalInterrupts](#) {
 [kExternalInterrupt0](#) , [kExternalInterrupt1](#) , [kExternalInterrupt2](#) , [kExternalInterrupt3](#) ,
 [kExternalInterrupt4](#) , [kExternalInterrupt5](#) , [kExternalInterrupt6](#) , [kExternalInterrupt7](#) ,
 [kExternalInterruptAll](#) }
This enum lists the external interrupts that can be suppressed (disabled). To pass more than one external interrupt, simply "or" them.
- enum [PinChangeInterrupts](#) { [kPinChangeInterrupt0](#) , [kPinChangeInterrupt1](#) , [kPinChangeInterrupt2](#) ,
 [kPinChangeInterruptAll](#) }
This enum lists the pin change interrupts that can be suppressed (disabled). To pass more than one pin change interrupt, simply "or" them.

8.3.1 Detailed Description

This namespace bundles various utility classes designed to suppress selected interrupts using the RAII idiom.

8.3.2 Enumeration Type Documentation

8.3.2.1 ExternalInterrupts

```
enum Interrupts::ExternalInterrupts
```

This enum lists the external interrupts that can be suppressed (disabled). To pass more than one external interrupt, simply "or" them.

Enumerator

kExternalInterrupt0	External interrupt 0.
kExternalInterrupt1	External interrupt 1.
kExternalInterrupt2	External interrupt 2 (ATmega2560 only)
kExternalInterrupt3	External interrupt 3 (ATmega2560 only)
kExternalInterrupt4	External interrupt 4 (ATmega2560 only)
kExternalInterrupt5	External interrupt 5 (ATmega2560 only)
kExternalInterrupt6	External interrupt 6 (ATmega2560 only)
kExternalInterrupt7	External interrupt 7 (ATmega2560 only)
kExternalInterruptAll	All external interrupts.

8.3.2.2 PinChangeInterrupts

```
enum Interrupts::PinChangeInterrupts
```

This enum lists the pin change interrupts that can be suppressed (disabled). To pass more than one pin change interrupt, simply "or" them.

Enumerator

kPinChangeInterrupt0	Pin change interrupt 0.
kPinChangeInterrupt1	Pin change interrupt 1.
kPinChangeInterrupt2	Pin change interrupt 2.
kPinChangeInterruptAll	All pin change interrupts.

8.4 MemUtils Namespace Reference

A namespace providing encapsulation for functions that report the available memory in SRAM.

Functions

- `size_t freeSRAM ()`
Get the total free memory remaining in SRAM.
- `size_t freeMemoryBetweenHeapAndStack ()`
Get the free memory between the heap and the stack.
- `void resetHeap ()`
Reset the heap to an empty (virgin) state.
- `size_t memoryAvailableOnFreeList ()`
Get the free memory on the heap free-list.
- `size_t getFreeListStats (int *nbrBlocks, size_t *sizeSmallestBlock, size_t *sizeLargestBlock)`
Get information about the heap free-list.

8.4.1 Detailed Description

A namespace providing encapsulation for functions that report the available memory in SRAM.

8.4.2 Function Documentation

8.4.2.1 `freeMemoryBetweenHeapAndStack()`

```
size_t MemUtils::freeMemoryBetweenHeapAndStack ( )
```

Get the free memory between the heap and the stack.

This does not include any memory potentially available within the heap on the free-list. It executes quickly, so this function is useful for checking to make sure the heap and stack aren't in danger of collision.

Returns

The number of free bytes remaining between the top of the heap and the top of the stack.

8.4.2.2 `freeSRAM()`

```
size_t MemUtils::freeSRAM ( )
```

Get the total free memory remaining in SRAM.

This includes memory on the free-list (if the heap is used) as well as memory available between the heap and the stack (which grow towards each other). If the heap has been used, the function will walk the free-list to determine the total amount of free memory.

Returns

The number of free bytes remaining in SRAM.

8.4.2.3 `getFreeListStats()`

```
size_t MemUtils::getFreeListStats (
    int * nbrBlocks,
    size_t * sizeSmallestBlock,
    size_t * sizeLargestBlock )
```

Get information about the heap free-list.

This provides information about the number of blocks on the free list, the size of the largest and smallest block, as well as the total memory held on the free list.

Note

This does NOT include any unallocated memory available between the top of the heap and the top of the stack.

- `nbrBlocks` returns the number of blocks in the free-list.
- `sizeSmallestBlock` returns the size of the smallest block on the free-list.
- `answerGeneralCall` returns the size of the largest block on the free-list.

Returns

The total number of free bytes on the heap's free list.

8.4.2.4 `memoryAvailableOnFreeList()`

```
size_t MemUtils::memoryAvailableOnFreeList ( )
```

Get the free memory on the heap free-list.

This shows the total free memory available on the free-list. This is the sum of the free-blocks contained within the heap.

Note

This does NOT include any unallocated memory available between the top of the heap and the top of the stack.

Returns

The total number of free bytes on the heap's free-list.

8.4.2.5 `resetHeap()`

```
void MemUtils::resetHeap ( )
```

Reset the heap to an empty (virgin) state.

This function resets the heap to an empty, pristine state. Not only are all memory allocations abandoned, but the free list is also purged, leaving everything between the start of the heap and the top of the stack as unallocated (actually, never-allocated) memory.

8.5 SPI Namespace Reference

This namespace bundles an interface to the SPI hardware subsystem on the AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega) microcontrollers. It provides logical cohesion for functions implement the Master portion of the SPI protocol and prevents namespace collisions.

Classes

- class [SPISettings](#)
A class that binds settings for configuring SPI transmissions.

Enumerations

- enum [ByteOrder](#) { [kLsbFirst](#) , [kMsbFirst](#) }
An enumeration that defines the byte order for multibyte SPI transmissions.
- enum [SpiMode](#) { [kSpiMode0](#) , [kSpiMode1](#) , [kSpiMode2](#) , [kSpiMode3](#) }
An enumeration that defines the modes available for SPI transmissions.

Functions

- void `enable` ()
Enable the SPI subsystem for transmission.
- void `disable` ()
Disable the SPI subsystem, precluding further transmissions.
- void `configure` (`SPISettings` settings)
Set the configuration of SPI subsystem to match the needs of the system you are going to communicate with.
- `uint8_t` `transmit` (`uint8_t` data)
Transmit a single byte using the SPI subsystem.
- `uint16_t` `transmit16` (`uint16_t` data)
Transmit a word-sized integer (two bytes) using the SPI subsystem. The order in which the bytes are sent is determined by the bit order configuration that has been set.
- `uint32_t` `transmit32` (`uint32_t` data)
Transmit a long-word-sized integer (four bytes) using the SPI subsystem. The order in which the bytes are sent is determined by the bit order configuration that has been set.
- void `transmit` (`uint8_t` *buffer, `size_t` count)
Transmit an array of bytes using the SPI subsystem. The bytes are transmitted in array order.

8.5.1 Detailed Description

This namespace bundles an interface to the SPI hardware subsystem on the AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega) microcontrollers. It provides logical cohesion for functions implement the Master portion of the SPI protocol and prevents namespace collisions.

These interfaces are synchronous, based on polling the flag in the SPI status register to determine transmission is complete and refill the transmit register with data. While it is possible to create an interrupt driven, asynchronous interface to the SPI subsystem, SPI-based communications are so fast that interrupt-based implementations are slower than polling by nearly a factor of 2. This is based on actual testing data which you can review [here] (https://www.tablix.org/~avian/blog/archives/2012/06/spi_interrupts_versus_polling/). What happens is that SPI can work at half the CPU frequency, which means the CPU can only execute about 16 instructions per byte sent. When the CPU is calling interrupts that often, the overhead of calling the interrupt function dominates, and is greater than the overhead of a simple polling loop.

The AVRTools implementation is based in part on the Arduino Library SPI module. In particular, the `SPISettings` class from the Arduino library is very cleverly and efficiently coded and has been adopted here. The lessons learned by the Arduino library SPI authors in correctly initializing the SPI subsystem have also been incorporated into this implementation. However, the packaging of the interface is somewhat different the AVRTools implementation takes a different approach to deconflicting SPI usage between the main thread of code execution and interrupt code.

The fundamental problem is this: if SPI is used in both the main code and in interrupt code, then it is important to ensure that the SPI "transactions" not be interleaved, that only one SPI "transaction" happen at a time. More specifically, you have to ensure that interrupt code using SPI does not interrupt an on-going SPI transaction in the main thread. The Arduino library achieves this by requiring library users to register any interrupts that use SPI and then requiring users to formally define (via function calls) the beginning and end of an SPI "transaction". The AVRTools library instead provides tools (via the `InterruptUtils` module) to temporarily suppress selected interrupts while the main thread is executing an SPI transaction. This approach allows more fine-tuned control of interrupt suppression, automatically restores interrupts when the SPI transaction is complete (no risk of a missing "end-of-transaction"), and does it with less overhead and memory footprint than the Arduino SPI library. The following code snippet illustrates how to protect main thread SPI usage from conflicts with SPI usage by two external interrupt handlers:

```
uint8_t send( uint8_t data )
```



```

{
    // SPI is used by the interrupt functions that respond to external interrupts 0 and 1,
    // so to prevent clashes, we suppress these two external interrupts for
    // the duration of this function
    Interrupts::ExternalOff interruptsOff( kExternalInterrupt0 | kExternalInterrupt1 );

    // Configure SPI
    SPI::configure( SPISettings( 4000000, SPI::kLsbFirst, SPI::kSpiMode2 ) );

    // Set the remote slave SS pin low to initiate a transmission
    setGpioPinLow( pConnectedToSlaveSSpin );

    // Transmit
    uint8_t retVal = SPI::transmit( data );

    // Set the remote slave SS pin high to terminate the transmission
    setGpioPinLow( pConnectedToSlaveSSpin );

    // Interrupts automatically reset when this function exits
    return retVal;
}

```

Note

This module implements SPI master mode only.

8.5.2 Enumeration Type Documentation

8.5.2.1 ByteOrder

```
enum SPI::ByteOrder
```

An enumeration that defines the byte order for multibyte SPI transmissions.

Enumerator

kLsbFirst	Least significant byte first.
kMsbFirst	Most significant byte first.

8.5.2.2 SpiMode

```
enum SPI::SpiMode
```

An enumeration that defines the modes available for SPI transmissions.

There are four modes controlling whether data is shifted in and out on the rising or falling edge of the data clock signal (called the phase, CPHA), and whether the clock is idle when high or low (called the polarity, CPOL). The four modes are simply the possible combinations of phase and polarity.

Enumerator

kSpiMode0	Phase falling, idle low (CPHA = 0, CPOL = 0)
kSpiMode1	Phase rising, idle low (CPHA = 1, CPOL = 0)
kSpiMode2	Phase falling, idle high (CPHA = 0, CPOL = 1)
kSpiMode3	Phase rising, idle high (CPHA = 1, CPOL = 1)

8.5.3 Function Documentation

8.5.3.1 `configure()`

```
void SPI::configure (
    SPISettings settings ) [inline]
```

Set the configuration of SPI subsystem to match the needs of the system you are going to communicate with.

You should always configure the SPI subsystem *before* transmitting any data. The configuration settings remain in place until a subsequent call to this function or until you disable SPI.

Note

If you are using SPI both from interrupts and from the main thread of execution, you must protect SPI onfigurations and transmissions from interleaving. To do this, disable interrupts in the main thread by using the appropriate objects from InterruptUtils. Interrupts should be disabled starting before setting the configuration until the end of the corresponding data transmission. For example:

```
uint8_t send( uint8_t data )
{
    // SPI is used by the interrupt functions that respond to external interrupts 0 and 1,
    // so to prevent clashes, we suppress these two external interrupts for
    // the duration of this function
    Interrupts::ExternalOff interruptsOff( kExternalInterrupt0 | kExternalInterrupt1 );

    // Configure SPI
    SPI::configure( SPISettings( 4000000, SPI::kLsbFirst, SPI::kSpiMode2 ) );

    // Set the remote slave SS pin low to initiate a transmission
    setGpioPinLow( pConnectedToSlaveSSpin );

    // Transmit
    uint8_t retVal = SPI::transmit( data );

    // Set the remote slave SS pin high to terminate the transmission
    setGpioPinLow( pConnectedToSlaveSSpin );

    // Interrupts automatically reset when this function exits
    return retVal;
}
```

8.5.3.2 `disable()`

```
void SPI::disable ( )
```

Disable the SPI subsystem, precluding further transmissions.

This call disables the SPI hardware, releasing the MOSI, MISO, CLK, and SS pins for other uses.

Note

No further SPI transmissions should be made after calling this function, unless you re-enable the SPI subsystem by again calling `enable()`.

8.5.3.3 enable()

```
void SPI::enable ( )
```

Enable the SPI subsystem for transmission.

This call enables the SPI hardware and configures the MOSI, MISO, CLK, and SS pins, making them unavailable for other uses. It also sets a default configuration of the SPI subsystem to a maximum transmission speed of 8 MHz, most significant bit first, and kSpiMode0.

Note

Even though SPI is configured in master-mode, the configuration of the SS pin is affected. The SS pin is set to output to prevent inadvertent automatic triggering of slave-mode by the SPI hardware (this happens if a low signal is received on the SS pin). Although the SS pin must be in output mode, it can still be used as a general purpose output port (it doesn't affect SPI operations as long as it remains in output mode).

8.5.3.4 transmit() [1/2]

```
void SPI::transmit (
    uint8_t * buffer,
    size_t count ) [inline]
```

Transmit an array of bytes using the SPI subsystem. The bytes are transmitted in array order.

- `buffer` the array of bytes to transmit. Incoming bytes are also stored here, replacing the outgoing data, byte-for-byte.

Returns

nothing, but the received stream of bytes is loaded into the buffer, replacing the data originally in the buffer.

8.5.3.5 transmit() [2/2]

```
uint8_t SPI::transmit (
    uint8_t data ) [inline]
```

Transmit a single byte using the SPI subsystem.

- `data` the byte to be transmitted.

Returns

the byte received from the SPI subsystem.

8.5.3.6 transmit16()

```
uint16_t SPI::transmit16 (
    uint16_t data ) [inline]
```

Transmit a word-sized integer (two bytes) using the SPI subsystem. The order in which the bytes are sent is determined by the bit order configuration that has been set.

- `data` the word-sized integer (two bytes) to be transmitted.

Returns

the word-sized integer (two bytes) received from the SPI subsystem, with byte order determined by the bit order configuration that has been set.

8.5.3.7 transmit32()

```
uint32_t SPI::transmit32 (
    uint32_t data )
```

Transmit a long-word-sized integer (four bytes) using the SPI subsystem. The order in which the bytes are sent is determined by the bit order configuration that has been set.

- `data` the long-word-sized integer (four bytes) to be transmitted.

Returns

the long-word-sized integer (four bytes) received from the SPI subsystem, with byte order determined by the bit order configuration that has been set.

8.6 USART0 Namespace Reference

This namespace bundles a high-level buffered interface to the USART0 hardware. It provides logical cohesion and prevents namespace collisions.

Functions

- void [start](#) (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Initialize USART0 for buffered, asynchronous serial communications using interrupts.
- void [stop](#) ()
Stops buffered serial communications using interrupts on USART0.
- size_t [write](#) (char c)
Write a single byte to the transmit buffer.
- size_t [write](#) (const char *c)
Write a null-terminated string to the transmit buffer.
- size_t [write](#) (const char *c, size_t n)
Write a character array of given size to the transmit buffer.
- size_t [write](#) (const uint8_t *c, size_t n)
Write a byte array of given size to the transmit buffer.
- void [flush](#) ()
Flush transmit buffer.
- int [peek](#) ()
Examine the next character in the receive buffer without removing it from the buffer.
- int [read](#) ()
Return the next character in the receive buffer, removing it from the buffer.
- bool [available](#) ()
Determine if there is data in the receive buffer..

8.6.1 Detailed Description

This namespace bundles a high-level buffered interface to the USART0 hardware. It provides logical cohesion and prevents namespace collisions.

8.6.2 Function Documentation

8.6.2.1 [available\(\)](#)

```
bool USART0::available ( )
```

Determine if there is data in the receive buffer..

Returns

if the receive buffer contains data, it returns TRUE; if the receive buffer is empty, it returns FALSE;

8.6.2.2 [flush\(\)](#)

```
void USART0::flush ( )
```

Flush transmit buffer.

This function blocks until the transmit buffer is empty and the last byte has been transmitted by USART0. [flush\(\)](#) doesn't actually do anything to make the transmit happen; it simply waits for the transmission to complete.

8.6.2.3 peek()

```
int USART0::peek ( )
```

Examine the next character in the receive buffer without removing it from the buffer.

Returns

if there is a value in the receive buffer, it returns the value (a number between 0 and 255); if the receive buffer is empty, it returns -1;

8.6.2.4 read()

```
int USART0::read ( )
```

Return the next character in the receive buffer, removing it from the buffer.

Returns

if there is a value in the receive buffer, it returns the value (a number between 0 and 255) and removes the value from the receive buffer; if the receive buffer is empty, it returns -1;

8.6.2.5 start()

```
void USART0::start (
    unsigned long baudRate,
    UartSerialConfiguration config = kSerial_8N1 )
```

Initialize USART0 for buffered, asynchronous serial communications using interrupts.

You must call this function before using any of the other [USART0](#) functions.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).
- `config` sets the configuration in term of data bits, parity, and stop bits. If omitted, the default is 8 data bits, no parity, and 1 stop bit.

8.6.2.6 stop()

```
void USART0::stop ( )
```

Stops buffered serial communications using interrupts on USART0.

After calling this function, Arduino pins 0 and 1 are released and available for use as ordinary digital pins.

If you want to use USART0 again for buffered, asynchronous serial communications, you must again call [start\(\)](#).

8.6.2.7 write() [1/4]

```
size_t USART0::write (
    char c )
```

Write a single byte to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART0-related interrupts.

- `c` the char (byte) to write into the transmit buffer

Returns

the number of bytes written into the output buffer.

8.6.2.8 write() [2/4]

```
size_t USART0::write (
    const char * c )
```

Write a null-terminated string to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART0-related interrupts.

- `c` the null-terminated string to write into the transmit buffer.

Returns

the number of bytes written into the output buffer.

8.6.2.9 write() [3/4]

```
size_t USART0::write (
    const char * c,
    size_t n )
```

Write a character array of given size to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART0-related interrupts

- `c` the character array to write into the transmit buffer.
- `n` the number of elements from the array to write into the transmit buffer.

Returns

the number of characters written into the output buffer.

8.6.2.10 write() [4/4]

```
size_t USART0::write (
    const uint8_t * c,
    size_t n )
```

Write a byte array of given size to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART0-related interrupts

- `c` the byte array to write into the transmit buffer.
- `n` the number of elements from the array to write into the transmit buffer.

Returns

the number of bytes written into the output buffer.

8.7 USART1 Namespace Reference

This namespace bundles a high-level buffered interface to the USART1 hardware. It provides logical cohesion and prevents namespace collisions.

Functions

- void [start](#) (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Initialize USART1 for buffered, asynchronous serial communications using interrupts.
- void [stop](#) ()
Stops buffered serial communications using interrupts on USART1.
- size_t [write](#) (char c)
Write a single byte to the transmit buffer.
- size_t [write](#) (const char *c)
Write a null-terminated string to the transmit buffer.
- size_t [write](#) (const char *c, size_t n)
Write a character array of given size to the transmit buffer.
- size_t [write](#) (const uint8_t *c, size_t n)
Write a byte array of given size to the transmit buffer.
- void [flush](#) ()
Flush transmit buffer.
- int [peek](#) ()
Examine the next character in the receive buffer without removing it from the buffer.
- int [read](#) ()
Return the next character in the receive buffer, removing it from the buffer.
- bool [available](#) ()
Determine if there is data in the receive buffer..

8.7.1 Detailed Description

This namespace bundles a high-level buffered interface to the USART1 hardware. It provides logical cohesion and prevents namespace collisions.

8.7.2 Function Documentation

8.7.2.1 `available()`

```
bool USART1::available ( )
```

Determine if there is data in the receive buffer..

Returns

if the receive buffer contains data, it returns TRUE; if the receive buffer is empty, it returns FALSE;

8.7.2.2 `flush()`

```
void USART1::flush ( )
```

Flush transmit buffer.

This function blocks until the transmit buffer is empty and the last byte has been transmitted by USART1. `flush()` doesn't actually do anything to make the transmit happen; it simply waits for the transmission to complete.

8.7.2.3 `peek()`

```
int USART1::peek ( )
```

Examine the next character in the receive buffer without removing it from the buffer.

Returns

if there is a value in the receive buffer, it returns the value (a number between 0 and 255); if the receive buffer is empty, it returns -1;

8.7.2.4 `read()`

```
int USART1::read ( )
```

Return the next character in the receive buffer, removing it from the buffer.

Returns

if there is a value in the receive buffer, it returns the value (a number between 0 and 255) and removes the value from the receive buffer; if the receive buffer is empty, it returns -1;

8.7.2.5 start()

```
void USART1::start (
    unsigned long baudRate,
    UsartSerialConfiguration config = kSerial_8N1 )
```

Initialize USART1 for buffered, asynchronous serial communications using interrupts.

You must call this function before using any of the other [USART1](#) functions.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).
- `config` sets the configuration in term of data bits, parity, and stop bits. If omitted, the default is 8 data bits, no parity, and 1 stop bit.

8.7.2.6 stop()

```
void USART1::stop ( )
```

Stops buffered serial communications using interrupts on USART1.

After calling this function, Arduino pins 0 and 1 are released and available for use as ordinary digital pins.

If you want to use USART1 again for buffered, asynchronous serial communications, you must again call [start\(\)](#).

8.7.2.7 write() [1/4]

```
size_t USART1::write (
    char c )
```

Write a single byte to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART1-related interrupts.

- `c` the char (byte) to write into the transmit buffer

Returns

the number of bytes written into the output buffer.

8.7.2.8 write() [2/4]

```
size_t USART1::write (
    const char * c )
```

Write a null-terminated string to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART1-related interrupts.

- `c` the null-terminated string to write into the transmit buffer.

Returns

the number of bytes written into the output buffer.

8.7.2.9 write() [3/4]

```
size_t USART1::write (
    const char * c,
    size_t n )
```

Write a character array of given size to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART1-related interrupts

- `c` the character array to write into the transmit buffer.
- `n` the number of elements from the array to write into the transmit buffer.

Returns

the number of characters written into the output buffer.

8.7.2.10 write() [4/4]

```
size_t USART1::write (
    const uint8_t * c,
    size_t n )
```

Write a byte array of given size to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART1-related interrupts

- `c` the byte array to write into the transmit buffer.
- `n` the number of elements from the array to write into the transmit buffer.

Returns

the number of bytes written into the output buffer.

8.8 USART2 Namespace Reference

This namespace bundles a high-level buffered interface to the USART2 hardware. It provides logical cohesion and prevents namespace collisions.

Functions

- void [start](#) (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Initialize USART2 for buffered, asynchronous serial communications using interrupts.
- void [stop](#) ()
Stops buffered serial communications using interrupts on USART2.
- size_t [write](#) (char c)
Write a single byte to the transmit buffer.
- size_t [write](#) (const char *c)
Write a null-terminated string to the transmit buffer.
- size_t [write](#) (const char *c, size_t n)
Write a character array of given size to the transmit buffer.
- size_t [write](#) (const uint8_t *c, size_t n)
Write a byte array of given size to the transmit buffer.
- void [flush](#) ()
Flush transmit buffer.
- int [peek](#) ()
Examine the next character in the receive buffer without removing it from the buffer.
- int [read](#) ()
Return the next character in the receive buffer, removing it from the buffer.
- bool [available](#) ()
Determine if there is data in the receive buffer..

8.8.1 Detailed Description

This namespace bundles a high-level buffered interface to the USART2 hardware. It provides logical cohesion and prevents namespace collisions.

8.8.2 Function Documentation

8.8.2.1 `available()`

```
bool USART2::available ( )
```

Determine if there is data in the receive buffer..

Returns

if the receive buffer contains data, it returns TRUE; if the receive buffer is empty, it returns FALSE;

8.8.2.2 `flush()`

```
void USART2::flush ( )
```

Flush transmit buffer.

This function blocks until the transmit buffer is empty and the last byte has been transmitted by USART2. `flush()` doesn't actually do anything to make the transmit happen; it simply waits for the transmission to complete.

8.8.2.3 `peek()`

```
int USART2::peek ( )
```

Examine the next character in the receive buffer without removing it from the buffer.

Returns

if there is a value in the receive buffer, it returns the value (a number between 0 and 255); if the receive buffer is empty, it returns -1;

8.8.2.4 `read()`

```
int USART2::read ( )
```

Return the next character in the receive buffer, removing it from the buffer.

Returns

if there is a value in the receive buffer, it returns the value (a number between 0 and 255) and removes the value from the receive buffer; if the receive buffer is empty, it returns -1;

8.8.2.5 start()

```
void USART2::start (
    unsigned long baudRate,
    UsartSerialConfiguration config = kSerial_8N1 )
```

Initialize USART2 for buffered, asynchronous serial communications using interrupts.

You must call this function before using any of the other [USART2](#) functions.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).
- `config` sets the configuration in term of data bits, parity, and stop bits. If omitted, the default is 8 data bits, no parity, and 1 stop bit.

8.8.2.6 stop()

```
void USART2::stop ( )
```

Stops buffered serial communications using interrupts on USART2.

After calling this function, Arduino pins 0 and 1 are released and available for use as ordinary digital pins.

If you want to use USART2 again for buffered, asynchronous serial communications, you must again call [start\(\)](#).

8.8.2.7 write() [1/4]

```
size_t USART2::write (
    char c )
```

Write a single byte to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART2-related interrupts.

- `c` the char (byte) to write into the transmit buffer

Returns

the number of bytes written into the output buffer.

8.8.2.8 write() [2/4]

```
size_t USART2::write (
    const char * c )
```

Write a null-terminated string to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART2-related interrupts.

- `c` the null-terminated string to write into the transmit buffer.

Returns

the number of bytes written into the output buffer.

8.8.2.9 write() [3/4]

```
size_t USART2::write (
    const char * c,
    size_t n )
```

Write a character array of given size to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART2-related interrupts

- `c` the character array to write into the transmit buffer.
- `n` the number of elements from the array to write into the transmit buffer.

Returns

the number of characters written into the output buffer.

8.8.2.10 write() [4/4]

```
size_t USART2::write (
    const uint8_t * c,
    size_t n )
```

Write a byte array of given size to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART2-related interrupts

- `c` the byte array to write into the transmit buffer.
- `n` the number of elements from the array to write into the transmit buffer.

Returns

the number of bytes written into the output buffer.

8.9 USART3 Namespace Reference

This namespace bundles a high-level buffered interface to the USART3 hardware. It provides logical cohesion and prevents namespace collisions.

Functions

- void [start](#) (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Initialize USART3 for buffered, asynchronous serial communications using interrupts.
- void [stop](#) ()
Stops buffered serial communications using interrupts on USART3.
- size_t [write](#) (char c)
Write a single byte to the transmit buffer.
- size_t [write](#) (const char *c)
Write a null-terminated string to the transmit buffer.
- size_t [write](#) (const char *c, size_t n)
Write a character array of given size to the transmit buffer.
- size_t [write](#) (const uint8_t *c, size_t n)
Write a byte array of given size to the transmit buffer.
- void [flush](#) ()
Flush transmit buffer.
- int [peek](#) ()
Examine the next character in the receive buffer without removing it from the buffer.
- int [read](#) ()
Return the next character in the receive buffer, removing it from the buffer.
- bool [available](#) ()
Determine if there is data in the receive buffer..

8.9.1 Detailed Description

This namespace bundles a high-level buffered interface to the USART3 hardware. It provides logical cohesion and prevents namespace collisions.

8.9.2 Function Documentation

8.9.2.1 `available()`

```
bool USART3::available ( )
```

Determine if there is data in the receive buffer..

Returns

if the receive buffer contains data, it returns TRUE; if the receive buffer is empty, it returns FALSE;

8.9.2.2 `flush()`

```
void USART3::flush ( )
```

Flush transmit buffer.

This function blocks until the transmit buffer is empty and the last byte has been transmitted by USART3. `flush()` doesn't actually do anything to make the transmit happen; it simply waits for the transmission to complete.

8.9.2.3 `peek()`

```
int USART3::peek ( )
```

Examine the next character in the receive buffer without removing it from the buffer.

Returns

if there is a value in the receive buffer, it returns the value (a number between 0 and 255); if the receive buffer is empty, it returns -1;

8.9.2.4 `read()`

```
int USART3::read ( )
```

Return the next character in the receive buffer, removing it from the buffer.

Returns

if there is a value in the receive buffer, it returns the value (a number between 0 and 255) and removes the value from the receive buffer; if the receive buffer is empty, it returns -1;

8.9.2.5 start()

```
void USART3::start (
    unsigned long baudRate,
    UsartSerialConfiguration config = kSerial_8N1 )
```

Initialize USART3 for buffered, asynchronous serial communications using interrupts.

You must call this function before using any of the other [USART3](#) functions.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).
- `config` sets the configuration in term of data bits, parity, and stop bits. If omitted, the default is 8 data bits, no parity, and 1 stop bit.

8.9.2.6 stop()

```
void USART3::stop ( )
```

Stops buffered serial communications using interrupts on USART3.

After calling this function, Arduino pins 0 and 1 are released and available for use as ordinary digital pins.

If you want to use USART3 again for buffered, asynchronous serial communications, you must again call [start\(\)](#).

8.9.2.7 write() [1/4]

```
size_t USART3::write (
    char c )
```

Write a single byte to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART3-related interrupts.

- `c` the char (byte) to write into the transmit buffer

Returns

the number of bytes written into the output buffer.

8.9.2.8 write() [2/4]

```
size_t USART3::write (
    const char * c )
```

Write a null-terminated string to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART3-related interrupts.

- `c` the null-terminated string to write into the transmit buffer.

Returns

the number of bytes written into the output buffer.

8.9.2.9 write() [3/4]

```
size_t USART3::write (
    const char * c,
    size_t n )
```

Write a character array of given size to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART3-related interrupts

- `c` the character array to write into the transmit buffer.
- `n` the number of elements from the array to write into the transmit buffer.

Returns

the number of characters written into the output buffer.

8.9.2.10 write() [4/4]

```
size_t USART3::write (
    const uint8_t * c,
    size_t n )
```

Write a byte array of given size to the transmit buffer.

This function attempts to queue the data into the transmit buffer. If there is room in the transmit buffer, the function returns immediately. If not, the function blocks waiting for room to become available in the transmit buffer.

The data is transmitted asynchronously via USART3-related interrupts

- `c` the byte array to write into the transmit buffer.
- `n` the number of elements from the array to write into the transmit buffer.

Returns

the number of bytes written into the output buffer.

Chapter 9

Class Documentation

9.1 Interrupts::AllOff Class Reference

This class defines an object that disables all interrupts during its lifetime. Interrupt state is restored by the object's destructor when the object goes out of scope.

```
#include <InterruptUtils.h>
```

Public Member Functions

- **AllOff ()**
Suppress all interrupts when the object is instantiated.
- **~AllOff ()**
Re-enable interrupts, restoring the interrupt state as it was when the object was instantiated.

9.1.1 Detailed Description

This class defines an object that disables all interrupts during its lifetime. Interrupt state is restored by the object's destructor when the object goes out of scope.

The documentation for this class was generated from the following file:

- [InterruptUtils.h](#)

9.2 Interrupts::ExternalOff Class Reference

This class defines an object that disables selected external interrupts during its lifetime. The selected external interrupts are restored by the object's destructor when it goes out of scope.

```
#include <InterruptUtils.h>
```

Public Member Functions

- [ExternalOff](#) (uint8_t whichOnesToTurnOff=kExternalInterruptMask)
Suppress some or all of the external interrupts when the object is instantiated.
- [~ExternalOff](#) ()
Re-enable the selected external interrupts.

9.2.1 Detailed Description

This class defines an object that disables selected external interrupts during its lifetime. The selected external interrupts are restored by the object's destructor when it goes out of scope.

9.2.2 Constructor & Destructor Documentation

9.2.2.1 ExternalOff()

```
Interrupts::ExternalOff::ExternalOff (
    uint8_t whichOnesToTurnOff = kExternalInterruptMask ) [inline]
```

Suppress some or all of the external interrupts when the object is instantiated.

- `whichOnesToTurnOff` is a bit mask, indicating the external interrupts to disable. The mask bits correspond to the bits in the External Interrupt Mask Register (EIMSK). If the argument is omitted, all external interrupts will be disabled.

The documentation for this class was generated from the following file:

- [InterruptUtils.h](#)

9.3 GpioPinVariable Class Reference

This class defines a type that can encode a GPIO pin as a variable. Read the section on [GPIO Pin Variables] ([GPIO pin variables](#)) to understand how to use this class.

```
#include <GpioPinMacros.h>
```

Public Member Functions

- Gpio8Ptr **ddr** () const
Return a pointer to the DDR register.
- Gpio8Ptr **port** () const
Return a pointer to the PORT register.
- Gpio8Ptr **pin** () const
Return a pointer to the PIN register.
- Gpio16Ptr **ocr** () const
Return a pointer to the OCR register (PWM related).
- Gpio8Ptr **tccr** () const
Return a pointer to the TCCR register (PWM related).
- uint8_t **bitNbr** () const
Return the bit number of this GPIO pin within the DDR, PORT, and PIN registers.
- uint8_t **com** () const
Return the bit number needed for manipulating TCCR register (PWM related).
- uint8_t **adcNbr** () const
Return the ADC channel number (analog-to-digital related).

9.3.1 Detailed Description

This class defines a type that can encode a GPIO pin as a variable. Read the section on [GPIO Pin Variables] ([GPIO pin variables](#)) to understand how to use this class.

There are also three macros that you need to create variables of type [GpioPinVariable](#): [makeGpioVarFromGpioPin\(\)](#), [makeGpioVarFromGpioPinAnalog\(\)](#), and [makeGpioVarFromGpioPinPwm\(\)](#). These are used like this:

```
GpioPinVariable pinA( makeGpioVarFromGpioPin( pPin10 ) );
GpioPinVariable pinB( makeGpioVarFromGpioPinAnalog( pPinA01 ) );
GpioPinVariable pinC = makeGpioVarFromGpioPinPwm( pPin03 );

GpioPinVariable pinArray[3];
pinArray[0] = pinA;
pinArray[1] = pinB;
pinArray[2] = makeGpioVarFromGpioPin( pPin07 );
```

Once you've done this, these variables can be assign and passed to functions as needed. To use these GPIO pin variables, there are special function analogs of the GPIO pin manipulation macros. These have the same names as the GPIO pin manipulation macros, except with a "V" appended.

The documentation for this class was generated from the following file:

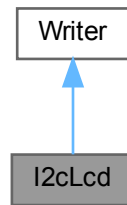
- [GpioPinMacros.h](#)

9.4 I2cLcd Class Reference

This class provides a high-level interface via I2C to an LCD such as those offered by AdaFruit and SparkFun. Specifically, it communicates via I2C with an MCP23017 that drives an HD44780U controlling an LCD. It also lets you detect button presses on the 5-button keypad generally associated with such devices.

```
#include <I2cLcd.h>
```

Inheritance diagram for I2cLcd:



Collaboration diagram for I2cLcd:



Public Types

- enum {
 kButton_Select , kButton_Right , kButton_Down , kButton_Up ,
 kButton_Left }
- enum {
 kBacklight_Red , kBacklight_Yellow , kBacklight_Green , kBacklight_Teal ,
 kBacklight_Blue , kBacklight_Violet , kBacklight_White }
- enum IntegerOutputBase { kBin = 2 , kOct = 8 , kDec = 10 , kHex = 16 }

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Public Member Functions

- **I2cLcd** ()
Constructor simply initializes some internal bookkeeping.
- int **init** ()
Initialize the [I2cLcd](#) object. This must be called before using the [I2cLcd](#), or calling any of the other member functions. The I2C system must be initialized before calling this function (by calling [I2cMaster::start\(\)](#) from [I2cMaster.h](#)).
- void **clear** ()
Clear the display (all rows, all columns).
- void **home** ()
Move the cursor home (the top row, left column).
- void **displayTopRow** (const char *str)
Display a C-string on the top row.
- void **displayBottomRow** (const char *str)
Display a C-string on the bottom row.
- void **clearTopRow** ()
Clear the top row.
- void **clearBottomRow** ()
Clear the bottom row.
- void **displayOff** ()
Turn the display off.
- void **displayOn** ()
Turn the display on.
- void **blinkOff** ()
Do not blink the cursor.
- void **blinkOn** ()
Blink the cursor.
- void **cursorOff** ()
Hide the cursor.
- void **cursorOn** ()
Display the cursor.
- void **scrollDisplayLeft** ()
Scroll the display to the left.
- void **scrollDisplayRight** ()
Scroll the display to the right.
- void **autoscrollOn** ()
Turn on automatic scrolling of the display.
- void **autoscrollOff** ()
Turn off automatic scrolling of the display.
- void **setCursor** (uint8_t row, uint8_t col)
Move the cursor the a particular row and column.
- int **setBacklight** (uint8_t color)
Set the backlight to a given color. Set a black-and-white LCD display to White if you want to have a backlight.
- void **command** (uint8_t cmd)
Pass a command to the LCD.
- uint8_t **readButtons** ()
Read the state of the buttons associated with the LCD display.

- virtual `size_t write (char c)`
Write a single character to the LCD at the current cursor location. This implements the pure virtual function `Writer::write(char c)`.
- virtual `size_t write (const char *str)`
Write a C-string to the LCD at the current cursor location. This implements the pure virtual function `Writer::write(const char str)`.*
- virtual `size_t write (const char *buffer, size_t size)`
Write a given number of characters from a buffer to the LCD at the current cursor location. This implements the pure virtual function `Writer::write(const char buffer, size_t size)`.*
- virtual `size_t write (const uint8_t *buffer, size_t size)`
Write a given number of bytes from a buffer to the LCD at the current cursor location. This implements the pure virtual function `Writer::write(const uint8_t buffer, size_t size)`.*
- virtual void **flush** ()
This function does nothing. It simply implements the pure virtual function `Writer::flush()`.
- `size_t print (const char *str, bool addLn=false)`
Print a null-terminated string to the output stream, with or without adding a new line character at the end.
- `size_t print (const uint8_t *buf, size_t size, bool addLn=false)`
Print a number of bytes to the output stream, with or without adding a new line character at the end.
- `size_t print (char c, bool addLn=false)`
Print a single character to the output stream, with or without adding a new line character at the end.
- `size_t print (int8_t n, int base=kDec, bool addLn=false)`
Print an 8-bit integer to the output stream, with or without adding a new line character at the end.
- `size_t print (uint8_t n, int base=kDec, bool addLn=false)`
Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.
- `size_t print (int n, int base=kDec, bool addLn=false)`
Print an integer to the output stream, with or without adding a new line character at the end.
- `size_t print (unsigned int n, int base=kDec, bool addLn=false)`
Print an unsigned integer to the output stream, with or without adding a new line character at the end.
- `size_t print (long n, int base=kDec, bool addLn=false)`
Print a long integer to the output stream, with or without adding a new line character at the end.
- `size_t print (unsigned long n, int base=kDec, bool addLn=false)`
Print an unsigned long integer to the output stream, with or without adding a new line character at the end.
- `size_t print (double d, int digits=2, bool addLn=false)`
Print a floating point number to the output stream, with or without adding a new line character at the end.
- `size_t println (const char *str)`
Print a null-terminated string to the output stream, adding a new line character at the end.
- `size_t println (const uint8_t *buf, size_t size)`
Print a number of bytes to the output stream, adding a new line character at the end.
- `size_t println (char c)`
Print a single character to the output stream, adding a new line character at the end.
- `size_t println (int8_t n, int base=kDec)`
Print an 8-bit integer to the output stream, adding a new line character at the end.
- `size_t println (uint8_t n, int base=kDec)`
Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (int n, int base=kDec)`
Print an integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned int n, int base=kDec)`
Print an unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (long n, int base=kDec)`

Print a long integer to the output stream, adding a new line character at the end.

- `size_t println` (unsigned long n, int base=`kDec`)

Print an unsigned long integer to the output stream, adding a new line character at the end.

- `size_t println` (double d, int digits=2)

Print a floating point number to the output stream, adding a new line character at the end.

- `size_t println` ()

Print a new line to the output stream.

9.4.1 Detailed Description

This class provides a high-level interface via I2C to an LCD such as those offered by AdaFruit and SparkFun. Specifically, it communicates via I2C with an MCP23017 that drives an HD44780U controlling an LCD. It also lets you detect button presses on the 5-button keypad generally associated with such devices.

This class derives from [Writer](#), allowing you to write to the LCD much as it if were a serial device.

To use these features, include [I2cLcd.h](#) in your source code and link against `I2cLcd.cpp` and `I2cMaster.cpp`, and initialize the I2C hardware by calling `I2cMaster::start()`.

9.4.2 Member Enumeration Documentation

9.4.2.1 anonymous enum

`anonymous enum`

These constants are used to identify the five buttons.

Enumerator

<code>kButton_Select</code>	the Select button
<code>kButton_Right</code>	the Right button
<code>kButton_Down</code>	the Down button
<code>kButton_Up</code>	the Up button
<code>kButton_Left</code>	the Left button

9.4.2.2 anonymous enum

`anonymous enum`

These constants are used to set the backlight color on the LCD.

Enumerator

<code>kBacklight_Red</code>	Backlight red.
-----------------------------	----------------

Enumerator

kBacklight_Yellow	Backlight yellow.
kBacklight_Green	Backlight green.
kBacklight_Teal	Backlight teal.
kBacklight_Blue	Backlight blue.
kBacklight_Violet	Backlight violet.
kBacklight_White	Backlight white.

9.4.2.3 IntegerOutputBase

```
enum Writer::IntegerOutputBase [inherited]
```

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Enumerator

kBin	Produce a binary representation of integers (e.g., 11 is output as 0b1011)
kOct	Produce an octal representation of integers (e.g, 11 is output as 013)
kDec	Produce a decimal representation of integers (e.g., 11 is output as 11.
kHex	Produce a hexadecimal representation of integers (e.g., 11 is output as 0x0b)

9.4.3 Member Function Documentation**9.4.3.1 command()**

```
void I2cLcd::command (
    uint8_t cmd )
```

Pass a command to the LCD.

- `cmd` a valid command to send to the HD44780U.

9.4.3.2 displayBottomRow()

```
void I2cLcd::displayBottomRow (
    const char * str )
```

Display a C-string on the bottom row.

- `str` the C-string to display.

9.4.3.3 displayTopRow()

```
void I2cLcd::displayTopRow (
    const char * str )
```

Display a C-string on the top row.

- `str` the C-string to display.

9.4.3.4 init()

```
int I2cLcd::init ( )
```

Initialize the [I2cLcd](#) object. This must be called before using the [I2cLcd](#), or calling any of the other member functions. The I2C system must be initialized before calling this function (by calling [I2cMaster::start\(\)](#) from [I2cMaster.h](#)).

The LCD display is initialized in 16-column, 2-row mode.

9.4.3.5 print() [1/10]

```
size_t Writer::print (
    char c,
    bool addLn = false ) [inherited]
```

Print a single character to the output stream, with or without adding a new line character at the end.

- `c` is the character to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.6 print() [2/10]

```
size_t Writer::print (
    const char * str,
    bool addLn = false ) [inherited]
```

Print a null-terminated string to the output stream, with or without adding a new line character at the end.

- `str` is the null-terminated string to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.7 print() [3/10]

```
size_t Writer::print (
    const uint8_t * buf,
    size_t size,
    bool addLn = false ) [inherited]
```

Print a number of bytes to the output stream, with or without adding a new line character at the end.

- `buf` is the buffer containing bytes to output.
- `size` is the number of bytes from the buffer to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.8 print() [4/10]

```
size_t Writer::print (
    double d,
    int digits = 2,
    bool addLn = false ) [inherited]
```

Print a floating point number to the output stream, with or without adding a new line character at the end.

- `d` is the floating point number to output.
- `digits` is the number of decimal digits to output; the default is 2.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.9 print() [5/10]

```
size_t Writer::print (
    int n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an integer to the output stream, with or without adding a new line character at the end.

- `n` is the integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.10 print() [6/10]

```
size_t Writer::print (
    int8_t n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an 8-bit integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.11 print() [7/10]

```
size_t Writer::print (
    long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print a long integer to the output stream, with or without adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.12 print() [8/10]

```
size_t Writer::print (
    uint8_t n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.13 print() [9/10]

```
size_t Writer::print (
    unsigned int n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.14 print() [10/10]

```
size_t Writer::print (
    unsigned long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an unsigned long integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.4.3.15 println() [1/10]

```
size_t Writer::println (
    char c ) [inline], [inherited]
```

Print a single character to the output stream, adding a new line character at the end.

- *c* is the character to output.

Returns

the number of bytes sent to the output stream.

9.4.3.16 println() [2/10]

```
size_t Writer::println (
    const char * str ) [inline], [inherited]
```

Print a null-terminated string to the output stream, adding a new line character at the end.

- *str* is the null-terminated string to output.

Returns

the number of bytes sent to the output stream.

9.4.3.17 `println()` [3/10]

```
size_t Writer::println (
    const uint8_t * buf,
    size_t size ) [inline], [inherited]
```

Print a number of bytes to the output stream, adding a new line character at the end.

- `buf` is the buffer containing bytes to output.
- `size` is the number of bytes from the buffer to output.

Returns

the number of bytes sent to the output stream.

9.4.3.18 `println()` [4/10]

```
size_t Writer::println (
    double d,
    int digits = 2 ) [inline], [inherited]
```

Print a floating point number to the output stream, adding a new line character at the end.

- `d` is the floating point number to output.
- `digits` is the number of decimal digits to output; the default is 2.

Returns

the number of bytes sent to the output stream.

9.4.3.19 `println()` [5/10]

```
size_t Writer::println (
    int n,
    int base = kDec ) [inline], [inherited]
```

Print an integer to the output stream, adding a new line character at the end.

- `n` is the integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.4.3.20 println() [6/10]

```
size_t Writer::println (
    int8_t n,
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit integer to the output stream, adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.4.3.21 println() [7/10]

```
size_t Writer::println (
    long n,
    int base = kDec ) [inline], [inherited]
```

Print a long integer to the output stream, adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.4.3.22 println() [8/10]

```
size_t Writer::println (
    uint8_t n,
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.4.3.23 println() [9/10]

```
size_t Writer::println (
    unsigned int n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned integer to the output stream, adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.4.3.24 println() [10/10]

```
size_t Writer::println (
    unsigned long n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned long integer to the output stream, adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.4.3.25 readButtons()

```
uint8_t I2cLcd::readButtons ( )
```

Read the state of the buttons associated with the LCD display.

Returns

a byte with flags set corresponding to the buttons that are depressed. You must "and" the return value with `kButton_Right`, `kButton_Left`, `kButton_Down`, `kButton_Up`, or `kButton_Select` to determine which buttons have been pressed.

9.4.3.26 setBacklight()

```
int I2cLcd::setBacklight (
    uint8_t color )
```

Set the backlight to a given color. Set a black-and-white LCD display to White if you want to have a backlight.

- `color` the color to set the backlight. Pass one of `kBacklight_Red`, `kBacklight_Yellow`, `kBacklight_Green`, `kBacklight_Teal`, `kBacklight_Blue`, `kBacklight_Violet`, or `kBacklight_White`.

9.4.3.27 setCursor()

```
void I2cLcd::setCursor (
    uint8_t row,
    uint8_t col )
```

Move the cursor the a particular row and column.

- `row` the row to move the cursor to (numbering starts at 0).
- `col` the column to move the cursor to (numbering starts at 0).

9.4.3.28 write() [1/4]

```
size_t I2cLcd::write (
    char c ) [virtual]
```

Write a single character to the LCD at the current cursor location. This implements the pure virtual function [Writer::write\(char c \)](#).

- the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.4.3.29 write() [2/4]

```
size_t I2cLcd::write (
    const char * buffer,
    size_t size ) [virtual]
```

Write a given number of characters from a buffer to the LCD at the current cursor location. This implements the pure virtual function [Writer::write\(const char* *buffer*, size_t *size* \)](#).

- *buffer* the buffer of characters to write.
- *size* the number of characters to write

Returns

the number of bytes written.

Implements [Writer](#).

9.4.3.30 write() [3/4]

```
size_t I2cLcd::write (
    const char * str ) [virtual]
```

Write a C-string to the LCD at the current cursor location. This implements the pure virtual function [Writer::write\(const char* *str* \)](#).

- the C-string to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.4.3.31 write() [4/4]

```
size_t I2cLcd::write (
    const uint8_t * buffer,
    size_t size ) [virtual]
```

Write a given number of bytes from a buffer to the LCD at the current cursor location. This implements the pure virtual function [Writer::write\(const uint8_t* *buffer*, size_t *size* \)](#).

- *buffer* the buffer of bytes to write.
- *size* the number of bytes to write

Returns

the number of bytes written.

Implements [Writer](#).

The documentation for this class was generated from the following files:

- [I2cLcd.h](#)
- [I2cLcd.cpp](#)

9.5 Interrupts::PinChangeOff Class Reference

This class defines an object that disables selected pin change interrupts during its lifetime. The selected pin change interrupts are restored by the object's destructor when it goes out of scope.

```
#include <InterruptUtils.h>
```

Public Member Functions

- [PinChangeOff](#) (uint8_t whichOnesToTurnOff=kPinChangeInterruptMask)
Suppress some or all of the pin change interrupts when the object is instantiated.
- [~PinChangeOff](#) ()
Re-enable the selected pin change interrupts.

9.5.1 Detailed Description

This class defines an object that disables selected pin change interrupts during its lifetime. The selected pin change interrupts are restored by the object's destructor when it goes out of scope.

9.5.2 Constructor & Destructor Documentation

9.5.2.1 PinChangeOff()

```
Interrupts::PinChangeOff::PinChangeOff (
    uint8_t whichOnesToTurnOff = kPinChangeInterruptMask ) [inline]
```

Suppress some or all of the pin change interrupts when the object is instantiated.

- `whichOnesToTurnOff` is a bit mask, indicating the pin change interrupts to disable. The mask bits correspond to the bits in the Pin Change Interrupt Control Register (PCICR). If the argument is omitted, all pin change interrupts will be disabled.

The documentation for this class was generated from the following file:

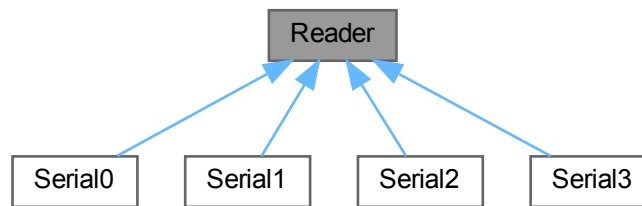
- [InterruptUtils.h](#)

9.6 Reader Class Reference

This is an abstract class defining a generic interface to read numbers and strings from a sequential stream of bytes (such as a serial device).

```
#include <Reader.h>
```

Inheritance diagram for Reader:



Public Member Functions

- **Reader ()**
Constructor. It sets the default timeout to 1 second.
- virtual int **read** ()=0
Pure virtual function that reads and removes the next byte from the input stream.
- virtual int **peek** ()=0
Pure virtual function that examines the next byte from the input stream, without removing it.
- virtual bool **available** ()=0
Pure virtual function that determines if data is available in the input stream.
- void **setTimeout** (unsigned long milliseconds)
Sets maximum milliseconds to wait for stream data, default is 1 second.
- bool **find** (const char *target)
Read data from the input stream until the target string is found.
- bool **find** (const char *target, size_t length)
Read data from the stream until the target string of given length is found.
- bool **findUntil** (const char *target, const char *terminator)
Read data from the stream until the target string is found, or the terminator string is found, or the function times out.
- bool **findUntil** (const char *target, size_t targetLen, const char *terminate, size_t termLen)
Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.
- bool **readLong** (long *result)
Return the first valid long integer value from the stream.
- bool **readFloat** (float *result)
Return the first valid float value from the stream.

- bool [readLong](#) (long *result, char skipChar)
Return the first valid long integer value from the stream, ignoring selected characters.
- bool [readFloat](#) (float *result, char skipChar)
Return the first valid float value from the stream, ignoring selected characters.
- size_t [readBytes](#) (char *buffer, size_t length)
Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is NOT null-terminated.
- size_t [readBytesUntil](#) (char terminator, char *buffer, size_t length)
Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is NOT null-terminated.
- size_t [readBytes](#) (uint8_t *buffer, size_t length)
Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.
- size_t [readBytesUntil](#) (uint8_t terminator, uint8_t *buffer, size_t length)
Read bytes (uint8_t) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.
- size_t [readLine](#) (char *buffer, size_t length)
Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result IS null-terminated.
- void [consumeWhiteSpace](#) ()
Consumes whitespace characters until the first non-whitespace character is encountered or the function times out.

9.6.1 Detailed Description

This is an abstract class defining a generic interface to read numbers and strings from a sequential stream of bytes (such as a serial device).

It implements functions to convert a sequence of bytes into various integers and floating point numbers (so it is not a pure interface class). These functions depend on a small set of lower-level functions that are purely abstract and must be implemented by classes deriving from [Reader](#).

[Serial0](#) is an example of a class that derives from [Reader](#) by implementing the purely abstract functions in [Reader](#).

Note

Use of the timeout feature requires linking against `SystemClock.cpp` and calling `initSystemClock()` from your start-up code. If you do not wish to use the system clock and link against `SystemClock.cpp`, then define the macro `USE_READER_WITHOUT_SYSTEM_CLOCK`. This means that calls will never timeout, and you are likely to lock your system if you read input that doesn't naturally terminate parsing (e.g., if you read numbers and the last number isn't followed by a newline).

9.6.2 Member Function Documentation

9.6.2.1 `available()`

```
virtual bool Reader::available ( ) [pure virtual]
```

Pure virtual function that determines if data is available in the input stream.

Returns

True if data is available in the stream before timeout expires; false if timeout expires before any data appears in the stream.

Implemented in [Serial0](#), [Serial1](#), [Serial2](#), and [Serial3](#).

9.6.2.2 find() [1/2]

```
bool Reader::find (
    const char * target ) [inline]
```

Read data from the input stream until the target string is found.

- `target` is the string the function seeks in the input stream.

Returns

true if target string is found before timeout, false otherwise.

9.6.2.3 find() [2/2]

```
bool Reader::find (
    const char * target,
    size_t length ) [inline]
```

Read data from the stream until the target string of given length is found.

- `target` is a string, the first length bytes of which the function seeks in the input stream.
- `length` is the number of bytes of the string to use for comparison.

Returns

true if target string of given length is found, false if the function times out before finding the target string.

9.6.2.4 findUntil() [1/2]

```
bool Reader::findUntil (
    const char * target,
    const char * terminator )
```

Read data from the stream until the target string is found, or the terminator string is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `terminator` is the string that stops the search.

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.6.2.5 findUntil() [2/2]

```
bool Reader::findUntil (
    const char * target,
    size_t targetLen,
    const char * terminate,
    size_t termLen )
```

Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `targetLen` is the number of bytes in target that the function seeks in the input stream.
- `terminator` is the string that stops the search.
- `termLen` is the number of bytes in the terminator that

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.6.2.6 peek()

```
virtual int Reader::peek ( ) [pure virtual]
```

Pure virtual function that examines the next byte from the input stream, without removing it.

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implemented in [Serial0](#), [Serial1](#), [Serial2](#), and [Serial3](#).

9.6.2.7 read()

```
virtual int Reader::read ( ) [pure virtual]
```

Pure virtual function that reads and removes the next byte from the input stream.

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implemented in [Serial0](#), [Serial1](#), [Serial2](#), and [Serial3](#).

9.6.2.8 readBytes() [1/2]

```
size_t Reader::readBytes (
    char * buffer,
    size_t length )
```

Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is *NOT* null-terminated.

- *buffer* a pointer to where the characters read will be stored.
- *length* the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout).

9.6.2.9 readBytes() [2/2]

```
size_t Reader::readBytes (
    uint8_t * buffer,
    size_t length ) [inline]
```

Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.

- *buffer* a pointer to where the bytes read will be stored.
- *length* the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout).

9.6.2.10 readBytesUntil() [1/2]

```
size_t Reader::readBytesUntil (
    char terminator,
    char * buffer,
    size_t length )
```

Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is *NOT* null-terminated.

- *terminator* a character that when encountered causes the function to return.
- *buffer* a pointer to where the characters read will be stored.
- *length* the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.6.2.11 readBytesUntil() [2/2]

```
size_t Reader::readBytesUntil (
    uint8_t terminator,
    uint8_t * buffer,
    size_t length ) [inline]
```

Read bytes (uint8_t) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.

- `terminator` a byte that when encountered causes the function to return.
- `buffer` a pointer to where the bytes read will be stored.
- `length` the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.6.2.12 readFloat() [1/2]

```
bool Reader::readFloat (
    float * result )
```

Return the first valid float value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit.

- `result` is a pointer to where the float will be stored.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.6.2.13 readFloat() [2/2]

```
bool Reader::readFloat (
    float * result,
    char skipChar )
```

Return the first valid float value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the float will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.6.2.14 `readLine()`

```
size_t Reader::readLine (
    char * buffer,
    size_t length )
```

Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result *IS* null-terminated.

- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting EOL).

9.6.2.15 `readLong()` [1/2]

```
bool Reader::readLong (
    long * result )
```

Return the first valid long integer value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit.

- `result` is a pointer to where the long integer will be stored.

Returns

true if a valid integer is found prior to timeout; false otherwise.

9.6.2.16 `readLong()` [2/2]

```
bool Reader::readLong (
    long * result,
    char skipChar )
```

Return the first valid long integer value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the long integer will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid long integer is found prior to timeout; false otherwise.

9.6.2.17 setTimeout()

```
void Reader::setTimeout (
    unsigned long milliseconds ) [inline]
```

Sets maximum milliseconds to wait for stream data, default is 1 second.

- `milliseconds` the length of the timeout period in milliseconds.

The documentation for this class was generated from the following files:

- [Reader.h](#)
- [Reader.cpp](#)

9.7 RingBuffer Class Reference

This class provides an efficient ring buffer implementation for storing bytes. Ring buffers are particularly useful for memory constrained microcontrollers such as the ATmega328 and ATmega2650. For maximum efficiency, this class is focused on the storage of bytes, providing a single code base that is shared by all instances of this class.

```
#include <RingBuffer.h>
```

Public Member Functions

- [RingBuffer](#) (unsigned char *buffer, unsigned short size)
Construct a ring buffer by providing the storage area for the ring buffer.
- int [pull](#) ()
Extract the next (first) byte from the ring buffer.
- int [peek](#) (unsigned short index=0)
Examine an element in the ring buffer.
- bool [push](#) (unsigned char element)
Push a byte into the ring buffer. The element is appended to the back of the buffer.
- bool [isFull](#) ()
Determine if the buffer is full and cannot accept more bytes.
- bool [isNotFull](#) ()
Determine if the buffer is not full and can accept more bytes.
- bool [isEmpty](#) ()
Determine if the buffer is empty .
- bool [isNotEmpty](#) ()
Determine if the buffer is not empty.
- void [clear](#) ()
Clear the ring buffer, leaving it empty.

9.7.1 Detailed Description

This class provides an efficient ring buffer implementation for storing bytes. Ring buffers are particularly useful for memory constrained microcontrollers such as the ATmega328 and ATmega2650. For maximum efficiency, this class is focused on the storage of bytes, providing a single code base that is shared by all instances of this class.

For maximum flexibility, the caller must provide the storage to be used for each [RingBuffer](#) object instantiated (this allows the use of different sized ring buffers without having to make dynamic memory allocations).

The implementation of [RingBuffer](#) is interrupt safe: the key operations are atomic, allowing for [RingBuffer](#) objects to be shared between interrupt functions and ordinary code.

The template-based [RingBufferT](#) class provides a more flexible ring buffer implementation that can store a variety of data types. However, this comes at the cost of replicating code for each template instantiation of [RingBufferT](#).

9.7.2 Constructor & Destructor Documentation

9.7.2.1 RingBuffer()

```
RingBuffer::RingBuffer (
    unsigned char * buffer,
    unsigned short size )
```

Construct a ring buffer by providing the storage area for the ring buffer.

- `buffer` the storage for the ring buffer.
- `size` the size of the storage for the ring buffer.

9.7.3 Member Function Documentation

9.7.3.1 isEmpty()

```
bool RingBuffer::isEmpty ( ) [inline]
```

Determine if the buffer is empty .

Returns

true if the buffer is empty; false if not.

9.7.3.2 isFull()

```
bool RingBuffer::isFull ( )
```

Determine if the buffer is full and cannot accept more bytes.

Returns

true if the buffer is full; false if not.

9.7.3.3 isEmpty()

```
bool RingBuffer::isEmpty ( ) [inline]
```

Determine if the buffer is not empty.

Returns

true if the buffer is not empty; false if it is empty.

9.7.3.4 isNotFull()

```
bool RingBuffer::isNotFull ( )
```

Determine if the buffer is not full and can accept more bytes.

Returns

true if the buffer is not full; false if it is full.

9.7.3.5 peek()

```
int RingBuffer::peek (
    unsigned short index = 0 )
```

Examine an element in the ring buffer.

- `index` the element to examine; 0 means the first (= next) element in the buffer. The default if the argument is omitted is to return the first element.

Returns

the next element or -1 if there is no such element.

9.7.3.6 pull()

```
int RingBuffer::pull ( )
```

Extract the next (first) byte from the ring buffer.

Returns

the next byte, or -1 if the ring buffer is empty.

9.7.3.7 push()

```
bool RingBuffer::push (
    unsigned char element )
```

Push a byte into the ring buffer. The element is appended to the back of the buffer.

- `element` is the byte to append to the ring buffer.

Returns

0 (false) if it succeeds; 1 (true) if it fails because the buffer is full.

The documentation for this class was generated from the following files:

- [RingBuffer.h](#)
- [RingBuffer.cpp](#)

9.8 RingBufferT< T, N, SIZE > Class Template Reference

a template-based ring buffer class that can store different kinds of objects in buffers of whatever size is needed.

```
#include <RingBufferT.h>
```

Public Member Functions

- **RingBufferT** ()
*Construct a ring buffer to store elements of type T indexed by integer type N, with size SIZE. All of these are passed as template parameters. *.*
- T **pull** ()
Extract the next (first) element from the ring buffer.
- T **peek** (N index=0)
Examine an element in the ring buffer.
- bool **push** (T element)
Push an element into the ring buffer. The element is appended to the back of the buffer.
- bool **isEmpty** ()
Determine if the buffer is empty .
- bool **isNotEmpty** ()
Determine if the buffer is not empty.
- bool **isFull** ()
Determine if the buffer is full and cannot accept more bytes.
- bool **isNotFull** ()
Determine if the buffer is not full and can accept more bytes.
- void **discardFromFront** (N nbrElements)
discard a number of elements from the front of the ring buffer.
- void **clear** ()
Clear the ring buffer, leaving it empty.

9.8.1 Detailed Description

```
template<typename T, typename N, unsigned int SIZE>
class RingBufferT< T, N, SIZE >
```

a template-based ring buffer class that can store different kinds of objects in buffers of whatever size is needed.

The implementation of [RingBufferT](#) is interrupt safe: the key operations are atomic, allowing for [RingBuffer](#) objects to be shared between interrupt functions and ordinary code.

The template-based [RingBufferT](#) class provides a very flexible ring buffer implementation; however different instantiations of [RingBufferT](#) (e.g., [RingBufferT](#)< char, int, 32 > and [RingBufferT](#)< char, int, 16 >) result in replicated code for each instantiation, even when they could logically share code. For a more efficient ring buffer that avoids such code bloat but can only store bytes, use [RingBuffer](#).

Template Parameters

<i>T</i>	is the type of object that will be stored in the RingBufferT instantiation.
<i>N</i>	is the integer type that will be used to index the RingBufferT elements.
<i>SIZE</i>	is an integer indicating the size of the RingBufferT instantiation.

9.8.2 Member Function Documentation

9.8.2.1 discardFromFront()

```
template<typename T , typename N , unsigned int SIZE>
void RingBufferT< T, N, SIZE >::discardFromFront (
    N nbrElements ) [inline]
```

discard a number of elements from the front of the ring buffer.

- *nbrElements* the number of elements to discard.

9.8.2.2 isEmpty()

```
template<typename T , typename N , unsigned int SIZE>
bool RingBufferT< T, N, SIZE >::isEmpty ( ) [inline]
```

Determine if the buffer is empty .

Returns

true if the buffer is empty; false if not.

9.8.2.3 isFull()

```
template<typename T , typename N , unsigned int SIZE>
bool RingBufferT< T, N, SIZE >::isFull ( ) [inline]
```

Determine if the buffer is full and cannot accept more bytes.

Returns

true if the buffer is full; false if not.

9.8.2.4 isEmpty()

```
template<typename T , typename N , unsigned int SIZE>
bool RingBufferT< T, N, SIZE >::isEmpty ( ) [inline]
```

Determine if the buffer is not empty.

Returns

true if the buffer is not empty; false if it is empty.

9.8.2.5 isNotFull()

```
template<typename T , typename N , unsigned int SIZE>
bool RingBufferT< T, N, SIZE >::isNotFull ( ) [inline]
```

Determine if the buffer is not full and can accept more bytes.

Returns

true if the buffer is not full; false if it is full.

9.8.2.6 peek()

```
template<typename T , typename N , unsigned int SIZE>
T RingBufferT< T, N, SIZE >::peek (
    N index = 0 ) [inline]
```

Examine an element in the ring buffer.

- `index` the element to examine; 0 means the first (= next) element in the buffer. The default if the argument is omitted is to return the first element.

Note

There is no general purpose safe value to return to indicate an empty element, so before calling `peek()` be sure the element exists.

Returns

the next element.

9.8.2.7 pull()

```
template<typename T , typename N , unsigned int SIZE>
T RingBufferT< T, N, SIZE >::pull ( ) [inline]
```

Extract the next (first) element from the ring buffer.

Note

There is no general purpose safe value to return to indicate an empty buffer, so before calling [pull\(\)](#) be sure to check the ring buffer is not empty.

Returns

the next element.

9.8.2.8 push()

```
template<typename T , typename N , unsigned int SIZE>
bool RingBufferT< T, N, SIZE >::push (
    T element ) [inline]
```

Push an element into the ring buffer. The element is appended to the back of the buffer.

- `element` is the item to append to the ring buffer.

Returns

0 (false) if it succeeds; 1 (true) if it fails because the buffer is full.

The documentation for this class was generated from the following file:

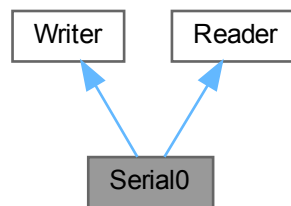
- [RingBufferT.h](#)

9.9 Serial0 Class Reference

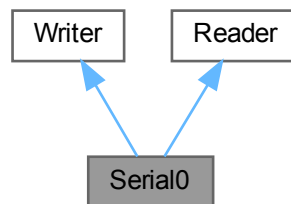
Provides a high-end interface to serial communications using USART0.

```
#include <USART0.h>
```

Inheritance diagram for Serial0:



Collaboration diagram for Serial0:



Public Types

- enum [IntegerOutputBase](#) { [kBin](#) = 2 , [kOct](#) = 8 , [kDec](#) = 10 , [kHex](#) = 16 }

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Public Member Functions

- void **start** (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial0](#) on USART0.
- void **stop** ()
Stops buffered serial communications using [Serial0](#) on USART0 by deconfiguring the hardware and turning off interrupts.
- virtual size_t **write** (char c)
Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).
- virtual size_t **write** (const char *str)
Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char str \)](#).*
- virtual size_t **write** (const char *buffer, size_t size)
Write a given number of characters from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const char buffer, size_t size \)](#).*
- virtual size_t **write** (const uint8_t *buffer, size_t size)
Write a given number of bytes from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const uint8_t buffer, size_t size \)](#).*
- virtual void **flush** ()
Flush the output stream. When this function returns, all previously written data will have been transmitted through the underlying output stream. This implements the pure virtual function [Writer::flush\(\)](#).
- virtual int **read** ()
Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).
- virtual int **peek** ()
Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).
- virtual bool **available** ()
Determine if data is available in the input stream. This implements the pure virtual function [Reader::available\(\)](#).
- size_t **print** (const char *str, bool addLn=false)
Print a null-terminated string to the output stream, with or without adding a new line character at the end.
- size_t **print** (const uint8_t *buf, size_t size, bool addLn=false)
Print a number of bytes to the output stream, with or without adding a new line character at the end.
- size_t **print** (char c, bool addLn=false)
Print a single character to the output stream, with or without adding a new line character at the end.
- size_t **print** (int8_t n, int base=[kDec](#), bool addLn=false)
Print an 8-bit integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (uint8_t n, int base=[kDec](#), bool addLn=false)
Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (int n, int base=[kDec](#), bool addLn=false)
Print an integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (unsigned int n, int base=[kDec](#), bool addLn=false)
Print an unsigned integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (long n, int base=[kDec](#), bool addLn=false)
Print a long integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (unsigned long n, int base=[kDec](#), bool addLn=false)
Print an unsigned long integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (double d, int digits=2, bool addLn=false)
Print a floating point number to the output stream, with or without adding a new line character at the end.
- size_t **println** (const char *str)
Print a null-terminated string to the output stream, adding a new line character at the end.

- `size_t println (const uint8_t *buf, size_t size)`
Print a number of bytes to the output stream, adding a new line character at the end.
- `size_t println (char c)`
Print a single character to the output stream, adding a new line character at the end.
- `size_t println (int8_t n, int base=kDec)`
Print an 8-bit integer to the output stream, adding a new line character at the end.
- `size_t println (uint8_t n, int base=kDec)`
Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (int n, int base=kDec)`
Print an integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned int n, int base=kDec)`
Print an unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (long n, int base=kDec)`
Print a long integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned long n, int base=kDec)`
Print an unsigned long integer to the output stream, adding a new line character at the end.
- `size_t println (double d, int digits=2)`
Print a floating point number to the output stream, adding a new line character at the end.
- `size_t println ()`
Print a new line to the output stream.
- `void setTimeout (unsigned long milliseconds)`
Sets maximum milliseconds to wait for stream data, default is 1 second.
- `bool find (const char *target)`
Read data from the input stream until the target string is found.
- `bool find (const char *target, size_t length)`
Read data from the stream until the target string of given length is found.
- `bool findUntil (const char *target, const char *terminator)`
Read data from the stream until the target string is found, or the terminator string is found, or the function times out.
- `bool findUntil (const char *target, size_t targetLen, const char *terminate, size_t termLen)`
Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.
- `bool readLong (long *result)`
Return the first valid long integer value from the stream.
- `bool readLong (long *result, char skipChar)`
Return the first valid long integer value from the stream, ignoring selected characters.
- `bool readFloat (float *result)`
Return the first valid float value from the stream.
- `bool readFloat (float *result, char skipChar)`
Return the first valid float value from the stream, ignoring selected characters.
- `size_t readBytes (char *buffer, size_t length)`
Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is NOT null-terminated.
- `size_t readBytes (uint8_t *buffer, size_t length)`
Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.
- `size_t readBytesUntil (char terminator, char *buffer, size_t length)`
Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is NOT null-terminated.

- `size_t readBytesUntil` (`uint8_t` terminator, `uint8_t *`buffer, `size_t` length)
Read bytes (`uint8_t`) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.
- `size_t readLine` (`char *`buffer, `size_t` length)
Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result IS null-terminated.
- `void consumeWhiteSpace` ()
Consumes whitespace characters until the first non-whitespace character is encountered or the function times out.

9.9.1 Detailed Description

Provides a high-end interface to serial communications using USART0.

The functions in this class are buffered for both input and output and operate using interrupts associated with USART0. This means the write functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART0 hardware. Similarly, data is received asynchronously and placed into the read buffer.

The read and write buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The read buffer, however, will overwrite if it gets full. You must clear the read buffer by actually reading the data regularly when receiving significant amounts of data.

9.9.2 Member Enumeration Documentation

9.9.2.1 IntegerOutputBase

```
enum Writer::IntegerOutputBase [inherited]
```

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Enumerator

kBin	Produce a binary representation of integers (e.g., 11 is output as 0b1011)
kOct	Produce an octal representation of integers (e.g, 11 is output as 013)
kDec	Produce a decimal representation of integers (e.g., 11 is output as 11.
kHex	Produce a hexadecimal representation of integers (e.g., 11 is output as 0x0b)

9.9.3 Member Function Documentation

9.9.3.1 available()

```
virtual bool Serial0::available ( ) [virtual]
```

Determine if data is available in the input stream. This implements the pure virtual function `Reader::available()`.

Returns

True if data is available in the stream; false if not.

Implements [Reader](#).

9.9.3.2 find() [1/2]

```
bool Reader::find (
    const char * target ) [inline], [inherited]
```

Read data from the input stream until the target string is found.

- `target` is the string the function seeks in the input stream.

Returns

true if target string is found before timeout, false otherwise.

9.9.3.3 find() [2/2]

```
bool Reader::find (
    const char * target,
    size_t length ) [inline], [inherited]
```

Read data from the stream until the target string of given length is found.

- `target` is a string, the first length bytes of which the function seeks in the input stream.
- `length` is the number of bytes of the string to use for comparison.

Returns

true if target string of given length is found, false if the function times out before finding the target string.

9.9.3.4 findUntil() [1/2]

```
bool Reader::findUntil (
    const char * target,
    const char * terminator ) [inherited]
```

Read data from the stream until the target string is found, or the terminator string is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `terminator` is the string that stops the search.

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.9.3.5 findUntil() [2/2]

```
bool Reader::findUntil (
    const char * target,
    size_t targetLen,
    const char * terminate,
    size_t termLen ) [inherited]
```

Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `targetLen` is the number of bytes in target that the function seeks in the input stream.
- `terminator` is the string that stops the search.
- `termLen` is the number of bytes in the terminator that

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.9.3.6 peek()

```
virtual int Serial0::peek ( ) [virtual]
```

Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implements [Reader](#).

9.9.3.7 print() [1/10]

```
size_t Writer::print (
    char c,
    bool addLn = false ) [inherited]
```

Print a single character to the output stream, with or without adding a new line character at the end.

- `c` is the character to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.8 print() [2/10]

```
size_t Writer::print (
    const char * str,
    bool addLn = false ) [inherited]
```

Print a null-terminated string to the output stream, with or without adding a new line character at the end.

- *str* is the null-terminated string to output.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.9 print() [3/10]

```
size_t Writer::print (
    const uint8_t * buf,
    size_t size,
    bool addLn = false ) [inherited]
```

Print a number of bytes to the output stream, with or without adding a new line character at the end.

- *buf* is the buffer containing bytes to output.
- *size* is the number of bytes from the buffer to output.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.10 print() [4/10]

```
size_t Writer::print (
    double d,
    int digits = 2,
    bool addLn = false ) [inherited]
```

Print a floating point number to the output stream, with or without adding a new line character at the end.

- *d* is the floating point number to output.
- *digits* is the number of decimal digits to output; the default is 2.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.11 print() [5/10]

```
size_t Writer::print (
    int n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.12 print() [6/10]

```
size_t Writer::print (
    int8_t n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an 8-bit integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.13 print() [7/10]

```
size_t Writer::print (
    long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print a long integer to the output stream, with or without adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.14 print() [8/10]

```
size_t Writer::print (
    uint8_t n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.15 print() [9/10]

```
size_t Writer::print (
    unsigned int n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an unsigned integer to the output stream, with or without adding a new line character at the end.

- `n` is the unsigned integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.16 print() [10/10]

```
size_t Writer::print (
    unsigned long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an unsigned long integer to the output stream, with or without adding a new line character at the end.

- `n` is the unsigned long integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.9.3.17 println() [1/10]

```
size_t Writer::println (
    char c ) [inline], [inherited]
```

Print a single character to the output stream, adding a new line character at the end.

- `c` is the character to output.

Returns

the number of bytes sent to the output stream.

9.9.3.18 `println()` [2/10]

```
size_t Writer::println (
    const char * str ) [inline], [inherited]
```

Print a null-terminated string to the output stream, adding a new line character at the end.

- `str` is the null-terminated string to output.

Returns

the number of bytes sent to the output stream.

9.9.3.19 `println()` [3/10]

```
size_t Writer::println (
    const uint8_t * buf,
    size_t size ) [inline], [inherited]
```

Print a number of bytes to the output stream, adding a new line character at the end.

- `buf` is the buffer containing bytes to output.
- `size` is the number of bytes from the buffer to output.

Returns

the number of bytes sent to the output stream.

9.9.3.20 `println()` [4/10]

```
size_t Writer::println (
    double d,
    int digits = 2 ) [inline], [inherited]
```

Print a floating point number to the output stream, adding a new line character at the end.

- `d` is the floating point number to output.
- `digits` is the number of decimal digits to output; the default is 2.

Returns

the number of bytes sent to the output stream.

9.9.3.21 println() [5/10]

```
size_t Writer::println (
    int n,
    int base = kDec ) [inline], [inherited]
```

Print an integer to the output stream, adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.9.3.22 println() [6/10]

```
size_t Writer::println (
    int8_t n,
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit integer to the output stream, adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.9.3.23 println() [7/10]

```
size_t Writer::println (
    long n,
    int base = kDec ) [inline], [inherited]
```

Print a long integer to the output stream, adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.9.3.24 `println()` [8/10]

```
size_t Writer::println (
    uint8_t n,
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.

- `n` is the unsigned integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.9.3.25 `println()` [9/10]

```
size_t Writer::println (
    unsigned int n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned integer to the output stream, adding a new line character at the end.

- `n` is the unsigned integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.9.3.26 `println()` [10/10]

```
size_t Writer::println (
    unsigned long n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned long integer to the output stream, adding a new line character at the end.

- `n` is the unsigned long integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.9.3.27 read()

```
virtual int Serial0::read ( ) [virtual]
```

Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implements [Reader](#).

9.9.3.28 readBytes() [1/2]

```
size_t Reader::readBytes (
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is *NOT* null-terminated.

- *buffer* a pointer to where the characters read will be stored.
- *length* the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout).

9.9.3.29 readBytes() [2/2]

```
size_t Reader::readBytes (
    uint8_t * buffer,
    size_t length ) [inline], [inherited]
```

Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.

- *buffer* a pointer to where the bytes read will be stored.
- *length* the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout).

9.9.3.30 readBytesUntil() [1/2]

```
size_t Reader::readBytesUntil (
    char terminator,
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is *NOT* null-terminated.

- `terminator` a character that when encountered causes the function to return.
- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.9.3.31 readBytesUntil() [2/2]

```
size_t Reader::readBytesUntil (
    uint8_t terminator,
    uint8_t * buffer,
    size_t length ) [inline], [inherited]
```

Read bytes (`uint8_t`) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.

- `terminator` a byte that when encountered causes the function to return.
- `buffer` a pointer to where the bytes read will be stored.
- `length` the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.9.3.32 readFloat() [1/2]

```
bool Reader::readFloat (
    float * result ) [inherited]
```

Return the first valid float value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit.

- `result` is a pointer to where the float will be stored.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.9.3.33 readFloat() [2/2]

```
bool Reader::readFloat (
    float * result,
    char skipChar ) [inherited]
```

Return the first valid float value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the float will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.9.3.34 readLine()

```
size_t Reader::readLine (
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result is null-terminated.

- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting EOL).

9.9.3.35 readLong() [1/2]

```
bool Reader::readLong (
    long * result )    [inherited]
```

Return the first valid long integer value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit.

- `result` is a pointer to where the long integer will be stored.

Returns

true if a valid integer is found prior to timeout; false otherwise.

9.9.3.36 readLong() [2/2]

```
bool Reader::readLong (
    long * result,
    char skipChar )    [inherited]
```

Return the first valid long integer value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the long integer will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid long integer is found prior to timeout; false otherwise.

9.9.3.37 setTimeout()

```
void Reader::setTimeout (
    unsigned long milliseconds )    [inline], [inherited]
```

Sets maximum milliseconds to wait for stream data, default is 1 second.

- `milliseconds` the length of the timeout period in milliseconds.

9.9.3.38 start()

```
void Serial0::start (
    unsigned long baudRate,
    UsartSerialConfiguration config = kSerial_8N1 ) [inline]
```

Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial0](#) on USART0.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).
- `config` sets the configuration in term of data bits, parity, and stop bits. If omitted, the default is 8 data bits, no parity, and 1 stop bit.

9.9.3.39 stop()

```
void Serial0::stop ( ) [inline]
```

Stops buffered serial communications using [Serial0](#) on USART0 by deconfiguring the hardware and turning off interrupts.

After calling this function, Arduino pins 0 and 1 are released and available for use as ordinary digital pins.

If you want to use [Serial0](#) again for communications, you must call [start\(\)](#) again.

9.9.3.40 write() [1/4]

```
virtual size_t Serial0::write (
    char c ) [virtual]
```

Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).

- `c` the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.9.3.41 write() [2/4]

```
virtual size_t Serial0::write (
    const char * buffer,
    size_t size ) [virtual]
```

Write a given number of characters from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const char* buffer, size_t size \)](#).

- `buffer` the buffer of characters to write.
- `size` the number of characters to write

Returns

the number of bytes written.

Implements [Writer](#).

9.9.3.42 write() [3/4]

```
virtual size_t Serial0::write (
    const char * str ) [virtual]
```

Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char* str \)](#).

- `str` the string to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.9.3.43 write() [4/4]

```
virtual size_t Serial0::write (
    const uint8_t * buffer,
    size_t size ) [virtual]
```

Write a given number of bytes from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const uint8_t* buffer, size_t size \)](#).

- `buffer` the buffer of bytes to write.
- `size` the number of bytes to write

Returns

the number of bytes written.

Implements [Writer](#).

The documentation for this class was generated from the following file:

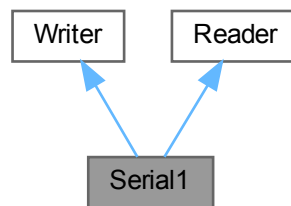
- [USART0.h](#)

9.10 Serial1 Class Reference

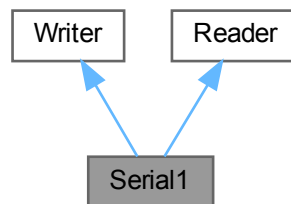
Provides a high-end interface to serial communications using USART1.

```
#include <USART1.h>
```

Inheritance diagram for Serial1:



Collaboration diagram for Serial1:



Public Types

- enum [IntegerOutputBase](#) { [kBin](#) = 2 , [kOct](#) = 8 , [kDec](#) = 10 , [kHex](#) = 16 }

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Public Member Functions

- void **start** (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial1](#) on USART1.
- void **stop** ()
Stops buffered serial communications using [Serial1](#) on USART1 by deconfiguring the hardware and turning off interrupts.
- virtual size_t **write** (char c)
Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).
- virtual size_t **write** (const char *str)
Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char str \)](#).*
- virtual size_t **write** (const char *buffer, size_t size)
Write a given number of characters from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const char buffer, size_t size \)](#).*
- virtual size_t **write** (const uint8_t *buffer, size_t size)
Write a given number of bytes from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const uint8_t buffer, size_t size \)](#).*
- virtual void **flush** ()
Flush the output stream. When this function returns, all previously written data will have been transmitted through the underlying output stream. This implements the pure virtual function [Writer::flush\(\)](#).
- virtual int **read** ()
Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).
- virtual int **peek** ()
Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).
- virtual bool **available** ()
Determine if data is available in the input stream. This implements the pure virtual function [Reader::available\(\)](#).
- size_t **print** (const char *str, bool addLn=false)
Print a null-terminated string to the output stream, with or without adding a new line character at the end.
- size_t **print** (const uint8_t *buf, size_t size, bool addLn=false)
Print a number of bytes to the output stream, with or without adding a new line character at the end.
- size_t **print** (char c, bool addLn=false)
Print a single character to the output stream, with or without adding a new line character at the end.
- size_t **print** (int8_t n, int base=[kDec](#), bool addLn=false)
Print an 8-bit integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (uint8_t n, int base=[kDec](#), bool addLn=false)
Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (int n, int base=[kDec](#), bool addLn=false)
Print an integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (unsigned int n, int base=[kDec](#), bool addLn=false)
Print an unsigned integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (long n, int base=[kDec](#), bool addLn=false)
Print a long integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (unsigned long n, int base=[kDec](#), bool addLn=false)
Print an unsigned long integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (double d, int digits=2, bool addLn=false)
Print a floating point number to the output stream, with or without adding a new line character at the end.
- size_t **println** (const char *str)
Print a null-terminated string to the output stream, adding a new line character at the end.

- `size_t println (const uint8_t *buf, size_t size)`
Print a number of bytes to the output stream, adding a new line character at the end.
- `size_t println (char c)`
Print a single character to the output stream, adding a new line character at the end.
- `size_t println (int8_t n, int base=kDec)`
Print an 8-bit integer to the output stream, adding a new line character at the end.
- `size_t println (uint8_t n, int base=kDec)`
Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (int n, int base=kDec)`
Print an integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned int n, int base=kDec)`
Print an unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (long n, int base=kDec)`
Print a long integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned long n, int base=kDec)`
Print an unsigned long integer to the output stream, adding a new line character at the end.
- `size_t println (double d, int digits=2)`
Print a floating point number to the output stream, adding a new line character at the end.
- `size_t println ()`
Print a new line to the output stream.
- `void setTimeout (unsigned long milliseconds)`
Sets maximum milliseconds to wait for stream data, default is 1 second.
- `bool find (const char *target)`
Read data from the input stream until the target string is found.
- `bool find (const char *target, size_t length)`
Read data from the stream until the target string of given length is found.
- `bool findUntil (const char *target, const char *terminator)`
Read data from the stream until the target string is found, or the terminator string is found, or the function times out.
- `bool findUntil (const char *target, size_t targetLen, const char *terminate, size_t termLen)`
Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.
- `bool readLong (long *result)`
Return the first valid long integer value from the stream.
- `bool readLong (long *result, char skipChar)`
Return the first valid long integer value from the stream, ignoring selected characters.
- `bool readFloat (float *result)`
Return the first valid float value from the stream.
- `bool readFloat (float *result, char skipChar)`
Return the first valid float value from the stream, ignoring selected characters.
- `size_t readBytes (char *buffer, size_t length)`
Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is NOT null-terminated.
- `size_t readBytes (uint8_t *buffer, size_t length)`
Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.
- `size_t readBytesUntil (char terminator, char *buffer, size_t length)`
Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is NOT null-terminated.

- `size_t readBytesUntil` (`uint8_t` terminator, `uint8_t *`buffer, `size_t` length)
Read bytes (`uint8_t`) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.
- `size_t readLine` (`char *`buffer, `size_t` length)
Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result IS null-terminated.
- `void consumeWhiteSpace` ()
Consumes whitespace characters until the first non-whitespace character is encountered or the function times out.

9.10.1 Detailed Description

Provides a high-end interface to serial communications using USART1.

The functions in this class are buffered for both input and output and operate using interrupts associated with USART1. This means the write functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART1 hardware. Similarly, data is received asynchronously and placed into the read buffer.

The read and write buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The read buffer, however, will overwrite if it gets full. You must clear the read buffer by actually reading the data regularly when receiving significant amounts of data.

9.10.2 Member Enumeration Documentation

9.10.2.1 IntegerOutputBase

```
enum Writer::IntegerOutputBase [inherited]
```

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Enumerator

kBin	Produce a binary representation of integers (e.g., 11 is output as 0b1011)
kOct	Produce an octal representation of integers (e.g, 11 is output as 013)
kDec	Produce a decimal representation of integers (e.g., 11 is output as 11.
kHex	Produce a hexadecimal representation of integers (e.g., 11 is output as 0x0b)

9.10.3 Member Function Documentation

9.10.3.1 available()

```
bool Serial1::available ( ) [virtual]
```

Determine if data is available in the input stream. This implements the pure virtual function [Reader::available\(\)](#).

Returns

True if data is available in the stream; false if not.

Implements [Reader](#).

9.10.3.2 find() [1/2]

```
bool Reader::find (
    const char * target ) [inline], [inherited]
```

Read data from the input stream until the target string is found.

- `target` is the string the function seeks in the input stream.

Returns

true if target string is found before timeout, false otherwise.

9.10.3.3 find() [2/2]

```
bool Reader::find (
    const char * target,
    size_t length ) [inline], [inherited]
```

Read data from the stream until the target string of given length is found.

- `target` is a string, the first length bytes of which the function seeks in the input stream.
- `length` is the number of bytes of the string to use for comparison.

Returns

true if target string of given length is found, false if the function times out before finding the target string.

9.10.3.4 findUntil() [1/2]

```
bool Reader::findUntil (
    const char * target,
    const char * terminator ) [inherited]
```

Read data from the stream until the target string is found, or the terminator string is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `terminator` is the string that stops the search.

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.10.3.5 findUntil() [2/2]

```
bool Reader::findUntil (
    const char * target,
    size_t targetLen,
    const char * terminate,
    size_t termLen ) [inherited]
```

Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `targetLen` is the number of bytes in `target` that the function seeks in the input stream.
- `terminator` is the string that stops the search.
- `termLen` is the number of bytes in the terminator that

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.10.3.6 peek()

```
int Serial1::peek ( ) [virtual]
```

Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implements [Reader](#).

9.10.3.7 print() [1/10]

```
size_t Writer::print (
    char c,
    bool addLn = false ) [inherited]
```

Print a single character to the output stream, with or without adding a new line character at the end.

- `c` is the character to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.8 print() [2/10]

```
size_t Writer::print (
    const char * str,
    bool addLn = false ) [inherited]
```

Print a null-terminated string to the output stream, with or without adding a new line character at the end.

- *str* is the null-terminated string to output.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.9 print() [3/10]

```
size_t Writer::print (
    const uint8_t * buf,
    size_t size,
    bool addLn = false ) [inherited]
```

Print a number of bytes to the output stream, with or without adding a new line character at the end.

- *buf* is the buffer containing bytes to output.
- *size* is the number of bytes from the buffer to output.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.10 print() [4/10]

```
size_t Writer::print (
    double d,
    int digits = 2,
    bool addLn = false ) [inherited]
```

Print a floating point number to the output stream, with or without adding a new line character at the end.

- *d* is the floating point number to output.
- *digits* is the number of decimal digits to output; the default is 2.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.11 print() [5/10]

```
size_t Writer::print (
    int n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.12 print() [6/10]

```
size_t Writer::print (
    int8_t n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an 8-bit integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.13 print() [7/10]

```
size_t Writer::print (
    long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print a long integer to the output stream, with or without adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.14 print() [8/10]

```
size_t Writer::print (
    uint8_t n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.15 print() [9/10]

```
size_t Writer::print (
    unsigned int n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.16 print() [10/10]

```
size_t Writer::print (
    unsigned long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an unsigned long integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.17 println() [1/10]

```
size_t Writer::println (
    char c ) [inline], [inherited]
```

Print a single character to the output stream, adding a new line character at the end.

- *c* is the character to output.

Returns

the number of bytes sent to the output stream.

9.10.3.18 println() [2/10]

```
size_t Writer::println (
    const char * str ) [inline], [inherited]
```

Print a null-terminated string to the output stream, adding a new line character at the end.

- `str` is the null-terminated string to output.

Returns

the number of bytes sent to the output stream.

9.10.3.19 println() [3/10]

```
size_t Writer::println (
    const uint8_t * buf,
    size_t size ) [inline], [inherited]
```

Print a number of bytes to the output stream, adding a new line character at the end.

- `buf` is the buffer containing bytes to output.
- `size` is the number of bytes from the buffer to output.

Returns

the number of bytes sent to the output stream.

9.10.3.20 println() [4/10]

```
size_t Writer::println (
    double d,
    int digits = 2 ) [inline], [inherited]
```

Print a floating point number to the output stream, adding a new line character at the end.

- `d` is the floating point number to output.
- `digits` is the number of decimal digits to output; the default is 2.

Returns

the number of bytes sent to the output stream.

9.10.3.21 println() [5/10]

```
size_t Writer::println (  
    int n,  
    int base = kDec ) [inline], [inherited]
```

Print an integer to the output stream, adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.10.3.22 println() [6/10]

```
size_t Writer::println (  
    int8_t n,  
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit integer to the output stream, adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.10.3.23 println() [7/10]

```
size_t Writer::println (  
    long n,  
    int base = kDec ) [inline], [inherited]
```

Print a long integer to the output stream, adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.10.3.24 println() [8/10]

```
size_t Writer::println (
    uint8_t n,
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.10.3.25 println() [9/10]

```
size_t Writer::println (
    unsigned int n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned integer to the output stream, adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.10.3.26 println() [10/10]

```
size_t Writer::println (
    unsigned long n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned long integer to the output stream, adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.10.3.27 read()

```
int Serial1::read ( ) [virtual]
```

Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implements [Reader](#).

9.10.3.28 readBytes() [1/2]

```
size_t Reader::readBytes (
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is *NOT* null-terminated.

- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout).

9.10.3.29 readBytes() [2/2]

```
size_t Reader::readBytes (
    uint8_t * buffer,
    size_t length ) [inline], [inherited]
```

Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.

- `buffer` a pointer to where the bytes read will be stored.
- `length` the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout).

9.10.3.30 readBytesUntil() [1/2]

```
size_t Reader::readBytesUntil (
    char terminator,
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is *NOT* null-terminated.

- `terminator` a character that when encountered causes the function to return.
- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.10.3.31 readBytesUntil() [2/2]

```
size_t Reader::readBytesUntil (
    uint8_t terminator,
    uint8_t * buffer,
    size_t length ) [inline], [inherited]
```

Read bytes (`uint8_t`) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.

- `terminator` a byte that when encountered causes the function to return.
- `buffer` a pointer to where the bytes read will be stored.
- `length` the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.10.3.32 readFloat() [1/2]

```
bool Reader::readFloat (
    float * result ) [inherited]
```

Return the first valid float value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit.

- `result` is a pointer to where the float will be stored.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.10.3.33 readFloat() [2/2]

```
bool Reader::readFloat (
    float * result,
    char skipChar ) [inherited]
```

Return the first valid float value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the float will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.10.3.34 readLine()

```
size_t Reader::readLine (
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result *IS* null-terminated.

- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting EOL).

9.10.3.35 readLong() [1/2]

```
bool Reader::readLong (
    long * result ) [inherited]
```

Return the first valid long integer value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit.

- `result` is a pointer to where the long integer will be stored.

Returns

true if a valid integer is found prior to timeout; false otherwise.

9.10.3.36 readLong() [2/2]

```
bool Reader::readLong (
    long * result,
    char skipChar ) [inherited]
```

Return the first valid long integer value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the long integer will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid long integer is found prior to timeout; false otherwise.

9.10.3.37 setTimeout()

```
void Reader::setTimeout (
    unsigned long milliseconds ) [inline], [inherited]
```

Sets maximum milliseconds to wait for stream data, default is 1 second.

- `milliseconds` the length of the timeout period in milliseconds.

9.10.3.38 start()

```
void Serial1::start (
    unsigned long baudRate,
    UsartSerialConfiguration config = kSerial_8N1 ) [inline]
```

Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial1](#) on USART1.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).
- `config` sets the configuration in term of data bits, parity, and stop bits. If omitted, the default is 8 data bits, no parity, and 1 stop bit.

9.10.3.39 stop()

```
void Serial1::stop ( ) [inline]
```

Stops buffered serial communications using [Serial1](#) on USART1 by deconfiguring the hardware and turning off interrupts.

After calling this function, Arduino pins 0 and 1 are released and available for use as ordinary digital pins.

If you want to use [Serial1](#) again for communications, you must call [start\(\)](#) again.

9.10.3.40 write() [1/4]

```
size_t Serial1::write (
    char c ) [virtual]
```

Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).

- `c` the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.10.3.41 write() [2/4]

```
size_t Serial1::write (
    const char * buffer,
    size_t size ) [virtual]
```

Write a given number of characters from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const char* *buffer*, size_t *size* \)](#).

- *buffer* the buffer of characters to write.
- *size* the number of characters to write

Returns

the number of bytes written.

Implements [Writer](#).

9.10.3.42 write() [3/4]

```
size_t Serial1::write (
    const char * str ) [virtual]
```

Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char* *str* \)](#).

- *str* the string to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.10.3.43 write() [4/4]

```
size_t Serial1::write (
    const uint8_t * buffer,
    size_t size ) [virtual]
```

Write a given number of bytes from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const uint8_t* *buffer*, size_t *size* \)](#).

- *buffer* the buffer of bytes to write.
- *size* the number of bytes to write

Returns

the number of bytes written.

Implements [Writer](#).

The documentation for this class was generated from the following files:

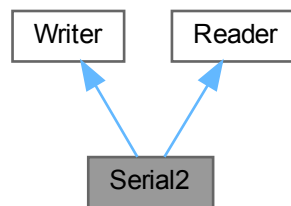
- [USART1.h](#)
- [USART1.cpp](#)

9.11 Serial2 Class Reference

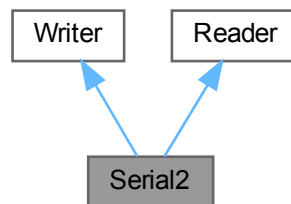
Provides a high-end interface to serial communications using USART2.

```
#include <USART2.h>
```

Inheritance diagram for Serial2:



Collaboration diagram for Serial2:



Public Types

- enum [IntegerOutputBase](#) { [kBin](#) = 2 , [kOct](#) = 8 , [kDec](#) = 10 , [kHex](#) = 16 }

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Public Member Functions

- void **start** (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial2](#) on USART2.
- void **stop** ()
Stops buffered serial communications using [Serial2](#) on USART2 by deconfiguring the hardware and turning off interrupts.
- virtual size_t **write** (char c)
Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).
- virtual size_t **write** (const char *str)
Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char str \)](#).*
- virtual size_t **write** (const char *buffer, size_t size)
Write a given number of characters from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const char buffer, size_t size \)](#).*
- virtual size_t **write** (const uint8_t *buffer, size_t size)
Write a given number of bytes from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const uint8_t buffer, size_t size \)](#).*
- virtual void **flush** ()
Flush the output stream. When this function returns, all previously written data will have been transmitted through the underlying output stream. This implements the pure virtual function [Writer::flush\(\)](#).
- virtual int **read** ()
Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).
- virtual int **peek** ()
Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).
- virtual bool **available** ()
Determine if data is available in the input stream. This implements the pure virtual function [Reader::available\(\)](#).
- size_t **print** (const char *str, bool addLn=false)
Print a null-terminated string to the output stream, with or without adding a new line character at the end.
- size_t **print** (const uint8_t *buf, size_t size, bool addLn=false)
Print a number of bytes to the output stream, with or without adding a new line character at the end.
- size_t **print** (char c, bool addLn=false)
Print a single character to the output stream, with or without adding a new line character at the end.
- size_t **print** (int8_t n, int base=[kDec](#), bool addLn=false)
Print an 8-bit integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (uint8_t n, int base=[kDec](#), bool addLn=false)
Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (int n, int base=[kDec](#), bool addLn=false)
Print an integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (unsigned int n, int base=[kDec](#), bool addLn=false)
Print an unsigned integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (long n, int base=[kDec](#), bool addLn=false)
Print a long integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (unsigned long n, int base=[kDec](#), bool addLn=false)
Print an unsigned long integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (double d, int digits=2, bool addLn=false)
Print a floating point number to the output stream, with or without adding a new line character at the end.
- size_t **println** (const char *str)
Print a null-terminated string to the output stream, adding a new line character at the end.

- `size_t println (const uint8_t *buf, size_t size)`
Print a number of bytes to the output stream, adding a new line character at the end.
- `size_t println (char c)`
Print a single character to the output stream, adding a new line character at the end.
- `size_t println (int8_t n, int base=kDec)`
Print an 8-bit integer to the output stream, adding a new line character at the end.
- `size_t println (uint8_t n, int base=kDec)`
Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (int n, int base=kDec)`
Print an integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned int n, int base=kDec)`
Print an unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (long n, int base=kDec)`
Print a long integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned long n, int base=kDec)`
Print an unsigned long integer to the output stream, adding a new line character at the end.
- `size_t println (double d, int digits=2)`
Print a floating point number to the output stream, adding a new line character at the end.
- `size_t println ()`
Print a new line to the output stream.
- `void setTimeout (unsigned long milliseconds)`
Sets maximum milliseconds to wait for stream data, default is 1 second.
- `bool find (const char *target)`
Read data from the input stream until the target string is found.
- `bool find (const char *target, size_t length)`
Read data from the stream until the target string of given length is found.
- `bool findUntil (const char *target, const char *terminator)`
Read data from the stream until the target string is found, or the terminator string is found, or the function times out.
- `bool findUntil (const char *target, size_t targetLen, const char *terminate, size_t termLen)`
Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.
- `bool readLong (long *result)`
Return the first valid long integer value from the stream.
- `bool readLong (long *result, char skipChar)`
Return the first valid long integer value from the stream, ignoring selected characters.
- `bool readFloat (float *result)`
Return the first valid float value from the stream.
- `bool readFloat (float *result, char skipChar)`
Return the first valid float value from the stream, ignoring selected characters.
- `size_t readBytes (char *buffer, size_t length)`
Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is NOT null-terminated.
- `size_t readBytes (uint8_t *buffer, size_t length)`
Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.
- `size_t readBytesUntil (char terminator, char *buffer, size_t length)`
Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is NOT null-terminated.

- `size_t readBytesUntil` (`uint8_t` terminator, `uint8_t *`buffer, `size_t` length)
Read bytes (`uint8_t`) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.
- `size_t readLine` (`char *`buffer, `size_t` length)
Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result IS null-terminated.
- `void consumeWhiteSpace` ()
Consumes whitespace characters until the first non-whitespace character is encountered or the function times out.

9.11.1 Detailed Description

Provides a high-end interface to serial communications using USART2.

The functions in this class are buffered for both input and output and operate using interrupts associated with USART2. This means the write functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART2 hardware. Similarly, data is received asynchronously and placed into the read buffer.

The read and write buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The read buffer, however, will overwrite if it gets full. You must clear the read buffer by actually reading the data regularly when receiving significant amounts of data.

9.11.2 Member Enumeration Documentation

9.11.2.1 IntegerOutputBase

```
enum Writer::IntegerOutputBase [inherited]
```

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Enumerator

kBin	Produce a binary representation of integers (e.g., 11 is output as 0b1011)
kOct	Produce an octal representation of integers (e.g, 11 is output as 013)
kDec	Produce a decimal representation of integers (e.g., 11 is output as 11.
kHex	Produce a hexadecimal representation of integers (e.g., 11 is output as 0x0b)

9.11.3 Member Function Documentation

9.11.3.1 available()

```
bool Serial2::available ( ) [virtual]
```

Determine if data is available in the input stream. This implements the pure virtual function `Reader::available()`.

Returns

True if data is available in the stream; false if not.

Implements [Reader](#).

9.11.3.2 find() [1/2]

```
bool Reader::find (
    const char * target ) [inline], [inherited]
```

Read data from the input stream until the target string is found.

- `target` is the string the function seeks in the input stream.

Returns

true if target string is found before timeout, false otherwise.

9.11.3.3 find() [2/2]

```
bool Reader::find (
    const char * target,
    size_t length ) [inline], [inherited]
```

Read data from the stream until the target string of given length is found.

- `target` is a string, the first length bytes of which the function seeks in the input stream.
- `length` is the number of bytes of the string to use for comparison.

Returns

true if target string of given length is found, false if the function times out before finding the target string.

9.11.3.4 findUntil() [1/2]

```
bool Reader::findUntil (
    const char * target,
    const char * terminator ) [inherited]
```

Read data from the stream until the target string is found, or the terminator string is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `terminator` is the string that stops the search.

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.11.3.5 findUntil() [2/2]

```
bool Reader::findUntil (
    const char * target,
    size_t targetLen,
    const char * terminate,
    size_t termLen ) [inherited]
```

Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `targetLen` is the number of bytes in target that the function seeks in the input stream.
- `terminator` is the string that stops the search.
- `termLen` is the number of bytes in the terminator that

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.11.3.6 peek()

```
int Serial2::peek ( ) [virtual]
```

Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implements [Reader](#).

9.11.3.7 print() [1/10]

```
size_t Writer::print (
    char c,
    bool addLn = false ) [inherited]
```

Print a single character to the output stream, with or without adding a new line character at the end.

- `c` is the character to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.8 print() [2/10]

```
size_t Writer::print (
    const char * str,
    bool addLn = false ) [inherited]
```

Print a null-terminated string to the output stream, with or without adding a new line character at the end.

- *str* is the null-terminated string to output.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.9 print() [3/10]

```
size_t Writer::print (
    const uint8_t * buf,
    size_t size,
    bool addLn = false ) [inherited]
```

Print a number of bytes to the output stream, with or without adding a new line character at the end.

- *buf* is the buffer containing bytes to output.
- *size* is the number of bytes from the buffer to output.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.10 print() [4/10]

```
size_t Writer::print (
    double d,
    int digits = 2,
    bool addLn = false ) [inherited]
```

Print a floating point number to the output stream, with or without adding a new line character at the end.

- *d* is the floating point number to output.
- *digits* is the number of decimal digits to output; the default is 2.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.11 print() [5/10]

```
size_t Writer::print (
    int n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.12 print() [6/10]

```
size_t Writer::print (
    int8_t n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an 8-bit integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.13 print() [7/10]

```
size_t Writer::print (
    long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print a long integer to the output stream, with or without adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.14 print() [8/10]

```
size_t Writer::print (
    uint8_t n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.15 print() [9/10]

```
size_t Writer::print (
    unsigned int n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.16 print() [10/10]

```
size_t Writer::print (
    unsigned long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an unsigned long integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.11.3.17 println() [1/10]

```
size_t Writer::println (
    char c ) [inline], [inherited]
```

Print a single character to the output stream, adding a new line character at the end.

- *c* is the character to output.

Returns

the number of bytes sent to the output stream.

9.11.3.18 `println()` [2/10]

```
size_t Writer::println (
    const char * str ) [inline], [inherited]
```

Print a null-terminated string to the output stream, adding a new line character at the end.

- `str` is the null-terminated string to output.

Returns

the number of bytes sent to the output stream.

9.11.3.19 `println()` [3/10]

```
size_t Writer::println (
    const uint8_t * buf,
    size_t size ) [inline], [inherited]
```

Print a number of bytes to the output stream, adding a new line character at the end.

- `buf` is the buffer containing bytes to output.
- `size` is the number of bytes from the buffer to output.

Returns

the number of bytes sent to the output stream.

9.11.3.20 `println()` [4/10]

```
size_t Writer::println (
    double d,
    int digits = 2 ) [inline], [inherited]
```

Print a floating point number to the output stream, adding a new line character at the end.

- `d` is the floating point number to output.
- `digits` is the number of decimal digits to output; the default is 2.

Returns

the number of bytes sent to the output stream.

9.11.3.21 println() [5/10]

```
size_t Writer::println (  
    int n,  
    int base = kDec ) [inline], [inherited]
```

Print an integer to the output stream, adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.11.3.22 println() [6/10]

```
size_t Writer::println (  
    int8_t n,  
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit integer to the output stream, adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.11.3.23 println() [7/10]

```
size_t Writer::println (  
    long n,  
    int base = kDec ) [inline], [inherited]
```

Print a long integer to the output stream, adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.11.3.24 println() [8/10]

```
size_t Writer::println (
    uint8_t n,
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.11.3.25 println() [9/10]

```
size_t Writer::println (
    unsigned int n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned integer to the output stream, adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.11.3.26 println() [10/10]

```
size_t Writer::println (
    unsigned long n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned long integer to the output stream, adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.11.3.27 read()

```
int Serial2::read ( ) [virtual]
```

Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implements [Reader](#).

9.11.3.28 readBytes() [1/2]

```
size_t Reader::readBytes (
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is *NOT* null-terminated.

- *buffer* a pointer to where the characters read will be stored.
- *length* the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout).

9.11.3.29 readBytes() [2/2]

```
size_t Reader::readBytes (
    uint8_t * buffer,
    size_t length ) [inline], [inherited]
```

Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.

- *buffer* a pointer to where the bytes read will be stored.
- *length* the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout).

9.11.3.30 readBytesUntil() [1/2]

```
size_t Reader::readBytesUntil (
    char terminator,
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is *NOT* null-terminated.

- `terminator` a character that when encountered causes the function to return.
- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.11.3.31 readBytesUntil() [2/2]

```
size_t Reader::readBytesUntil (
    uint8_t terminator,
    uint8_t * buffer,
    size_t length ) [inline], [inherited]
```

Read bytes (`uint8_t`) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.

- `terminator` a byte that when encountered causes the function to return.
- `buffer` a pointer to where the bytes read will be stored.
- `length` the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.11.3.32 readFloat() [1/2]

```
bool Reader::readFloat (
    float * result ) [inherited]
```

Return the first valid float value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit.

- `result` is a pointer to where the float will be stored.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.11.3.33 readFloat() [2/2]

```
bool Reader::readFloat (
    float * result,
    char skipChar ) [inherited]
```

Return the first valid float value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the float will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.11.3.34 readLine()

```
size_t Reader::readLine (
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result *IS* null-terminated.

- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting EOL).

9.11.3.35 readLong() [1/2]

```
bool Reader::readLong (
    long * result )    [inherited]
```

Return the first valid long integer value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit.

- `result` is a pointer to where the long integer will be stored.

Returns

true if a valid integer is found prior to timeout; false otherwise.

9.11.3.36 readLong() [2/2]

```
bool Reader::readLong (
    long * result,
    char skipChar )    [inherited]
```

Return the first valid long integer value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the long integer will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid long integer is found prior to timeout; false otherwise.

9.11.3.37 setTimeout()

```
void Reader::setTimeout (
    unsigned long milliseconds )    [inline], [inherited]
```

Sets maximum milliseconds to wait for stream data, default is 1 second.

- `milliseconds` the length of the timeout period in milliseconds.

9.11.3.38 start()

```
void Serial2::start (
    unsigned long baudRate,
    UsartSerialConfiguration config = kSerial_8N1 ) [inline]
```

Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial2](#) on USART2.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).
- `config` sets the configuration in term of data bits, parity, and stop bits. If omitted, the default is 8 data bits, no parity, and 1 stop bit.

9.11.3.39 stop()

```
void Serial2::stop ( ) [inline]
```

Stops buffered serial communications using [Serial2](#) on USART2 by deconfiguring the hardware and turning off interrupts.

After calling this function, Arduino pins 0 and 1 are released and available for use as ordinary digital pins.

If you want to use [Serial2](#) again for communications, you must call [start\(\)](#) again.

9.11.3.40 write() [1/4]

```
size_t Serial2::write (
    char c ) [virtual]
```

Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).

- `c` the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.11.3.41 write() [2/4]

```
size_t Serial2::write (  
    const char * buffer,  
    size_t size ) [virtual]
```

Write a given number of characters from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const char* buffer, size_t size \)](#).

- `buffer` the buffer of characters to write.
- `size` the number of characters to write

Returns

the number of bytes written.

Implements [Writer](#).

9.11.3.42 write() [3/4]

```
size_t Serial2::write (  
    const char * str ) [virtual]
```

Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char* str \)](#).

- `str` the string to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.11.3.43 write() [4/4]

```
size_t Serial2::write (  
    const uint8_t * buffer,  
    size_t size ) [virtual]
```

Write a given number of bytes from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const uint8_t* buffer, size_t size \)](#).

- `buffer` the buffer of bytes to write.
- `size` the number of bytes to write

Returns

the number of bytes written.

Implements [Writer](#).

The documentation for this class was generated from the following files:

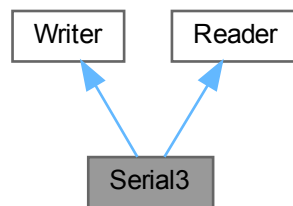
- [USART2.h](#)
- [USART2.cpp](#)

9.12 Serial3 Class Reference

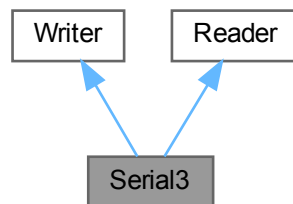
Provides a high-end interface to serial communications using USART3.

```
#include <USART3.h>
```

Inheritance diagram for Serial3:



Collaboration diagram for Serial3:



Public Types

- enum [IntegerOutputBase](#) { [kBin](#) = 2 , [kOct](#) = 8 , [kDec](#) = 10 , [kHex](#) = 16 }

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Public Member Functions

- void **start** (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial3](#) on USART3.
- void **stop** ()
Stops buffered serial communications using [Serial3](#) on USART3 by deconfiguring the hardware and turning off interrupts.
- virtual size_t **write** (char c)
Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).
- virtual size_t **write** (const char *str)
Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char str \)](#).*
- virtual size_t **write** (const char *buffer, size_t size)
Write a given number of characters from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const char buffer, size_t size \)](#).*
- virtual size_t **write** (const uint8_t *buffer, size_t size)
Write a given number of bytes from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const uint8_t buffer, size_t size \)](#).*
- virtual void **flush** ()
Flush the output stream. When this function returns, all previously written data will have been transmitted through the underlying output stream. This implements the pure virtual function [Writer::flush\(\)](#).
- virtual int **read** ()
Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).
- virtual int **peek** ()
Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).
- virtual bool **available** ()
Determine if data is available in the input stream. This implements the pure virtual function [Reader::available\(\)](#).
- size_t **print** (const char *str, bool addLn=false)
Print a null-terminated string to the output stream, with or without adding a new line character at the end.
- size_t **print** (const uint8_t *buf, size_t size, bool addLn=false)
Print a number of bytes to the output stream, with or without adding a new line character at the end.
- size_t **print** (char c, bool addLn=false)
Print a single character to the output stream, with or without adding a new line character at the end.
- size_t **print** (int8_t n, int base=[kDec](#), bool addLn=false)
Print an 8-bit integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (uint8_t n, int base=[kDec](#), bool addLn=false)
Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (int n, int base=[kDec](#), bool addLn=false)
Print an integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (unsigned int n, int base=[kDec](#), bool addLn=false)
Print an unsigned integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (long n, int base=[kDec](#), bool addLn=false)
Print a long integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (unsigned long n, int base=[kDec](#), bool addLn=false)
Print an unsigned long integer to the output stream, with or without adding a new line character at the end.
- size_t **print** (double d, int digits=2, bool addLn=false)
Print a floating point number to the output stream, with or without adding a new line character at the end.
- size_t **println** (const char *str)
Print a null-terminated string to the output stream, adding a new line character at the end.

- `size_t println (const uint8_t *buf, size_t size)`
Print a number of bytes to the output stream, adding a new line character at the end.
- `size_t println (char c)`
Print a single character to the output stream, adding a new line character at the end.
- `size_t println (int8_t n, int base=kDec)`
Print an 8-bit integer to the output stream, adding a new line character at the end.
- `size_t println (uint8_t n, int base=kDec)`
Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (int n, int base=kDec)`
Print an integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned int n, int base=kDec)`
Print an unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (long n, int base=kDec)`
Print a long integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned long n, int base=kDec)`
Print an unsigned long integer to the output stream, adding a new line character at the end.
- `size_t println (double d, int digits=2)`
Print a floating point number to the output stream, adding a new line character at the end.
- `size_t println ()`
Print a new line to the output stream.
- `void setTimeout (unsigned long milliseconds)`
Sets maximum milliseconds to wait for stream data, default is 1 second.
- `bool find (const char *target)`
Read data from the input stream until the target string is found.
- `bool find (const char *target, size_t length)`
Read data from the stream until the target string of given length is found.
- `bool findUntil (const char *target, const char *terminator)`
Read data from the stream until the target string is found, or the terminator string is found, or the function times out.
- `bool findUntil (const char *target, size_t targetLen, const char *terminate, size_t termLen)`
Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.
- `bool readLong (long *result)`
Return the first valid long integer value from the stream.
- `bool readLong (long *result, char skipChar)`
Return the first valid long integer value from the stream, ignoring selected characters.
- `bool readFloat (float *result)`
Return the first valid float value from the stream.
- `bool readFloat (float *result, char skipChar)`
Return the first valid float value from the stream, ignoring selected characters.
- `size_t readBytes (char *buffer, size_t length)`
Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is NOT null-terminated.
- `size_t readBytes (uint8_t *buffer, size_t length)`
Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.
- `size_t readBytesUntil (char terminator, char *buffer, size_t length)`
Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is NOT null-terminated.

- `size_t readBytesUntil` (`uint8_t` terminator, `uint8_t *`buffer, `size_t` length)
Read bytes (`uint8_t`) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.
- `size_t readLine` (`char *`buffer, `size_t` length)
Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result IS null-terminated.
- `void consumeWhiteSpace` ()
Consumes whitespace characters until the first non-whitespace character is encountered or the function times out.

9.12.1 Detailed Description

Provides a high-end interface to serial communications using USART3.

The functions in this class are buffered for both input and output and operate using interrupts associated with USART3. This means the write functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART3 hardware. Similarly, data is received asynchronously and placed into the read buffer.

The read and write buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The read buffer, however, will overwrite if it gets full. You must clear the read buffer by actually reading the data regularly when receiving significant amounts of data.

9.12.2 Member Enumeration Documentation

9.12.2.1 IntegerOutputBase

```
enum Writer::IntegerOutputBase [inherited]
```

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Enumerator

<code>kBin</code>	Produce a binary representation of integers (e.g., 11 is output as 0b1011)
<code>kOct</code>	Produce an octal representation of integers (e.g, 11 is output as 013)
<code>kDec</code>	Produce a decimal representation of integers (e.g., 11 is output as 11.
<code>kHex</code>	Produce a hexadecimal representation of integers (e.g., 11 is output as 0x0b)

9.12.3 Member Function Documentation

9.12.3.1 available()

```
bool Serial3::available ( ) [virtual]
```

Determine if data is available in the input stream. This implements the pure virtual function `Reader::available()`.

Returns

True if data is available in the stream; false if not.

Implements [Reader](#).

9.12.3.2 find() [1/2]

```
bool Reader::find (
    const char * target ) [inline], [inherited]
```

Read data from the input stream until the target string is found.

- `target` is the string the function seeks in the input stream.

Returns

true if target string is found before timeout, false otherwise.

9.12.3.3 find() [2/2]

```
bool Reader::find (
    const char * target,
    size_t length ) [inline], [inherited]
```

Read data from the stream until the target string of given length is found.

- `target` is a string, the first length bytes of which the function seeks in the input stream.
- `length` is the number of bytes of the string to use for comparison.

Returns

true if target string of given length is found, false if the function times out before finding the target string.

9.12.3.4 findUntil() [1/2]

```
bool Reader::findUntil (
    const char * target,
    const char * terminator ) [inherited]
```

Read data from the stream until the target string is found, or the terminator string is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `terminator` is the string that stops the search.

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.12.3.5 findUntil() [2/2]

```
bool Reader::findUntil (
    const char * target,
    size_t targetLen,
    const char * terminate,
    size_t termLen ) [inherited]
```

Read data from the stream until the target string of given length is found, or the terminator string of given length is found, or the function times out.

This function is like [find\(\)](#) but the search ends if the terminator string is found first.

- `target` is the string the function seeks in the input stream.
- `targetLen` is the number of bytes in target that the function seeks in the input stream.
- `terminator` is the string that stops the search.
- `termLen` is the number of bytes in the terminator that

Returns

true if target string is found before the terminator is encountered and before the function times out; false otherwise.

9.12.3.6 peek()

```
int Serial3::peek ( ) [virtual]
```

Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implements [Reader](#).

9.12.3.7 print() [1/10]

```
size_t Writer::print (
    char c,
    bool addLn = false ) [inherited]
```

Print a single character to the output stream, with or without adding a new line character at the end.

- `c` is the character to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.8 print() [2/10]

```
size_t Writer::print (
    const char * str,
    bool addLn = false ) [inherited]
```

Print a null-terminated string to the output stream, with or without adding a new line character at the end.

- *str* is the null-terminated string to output.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.9 print() [3/10]

```
size_t Writer::print (
    const uint8_t * buf,
    size_t size,
    bool addLn = false ) [inherited]
```

Print a number of bytes to the output stream, with or without adding a new line character at the end.

- *buf* is the buffer containing bytes to output.
- *size* is the number of bytes from the buffer to output.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.10 print() [4/10]

```
size_t Writer::print (
    double d,
    int digits = 2,
    bool addLn = false ) [inherited]
```

Print a floating point number to the output stream, with or without adding a new line character at the end.

- *d* is the floating point number to output.
- *digits* is the number of decimal digits to output; the default is 2.
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.11 print() [5/10]

```
size_t Writer::print (
    int n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.12 print() [6/10]

```
size_t Writer::print (
    int8_t n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an 8-bit integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.13 print() [7/10]

```
size_t Writer::print (
    long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print a long integer to the output stream, with or without adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.14 print() [8/10]

```
size_t Writer::print (
    uint8_t n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.15 print() [9/10]

```
size_t Writer::print (
    unsigned int n,
    int base = kDec,
    bool addLn = false ) [inline], [inherited]
```

Print an unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.16 print() [10/10]

```
size_t Writer::print (
    unsigned long n,
    int base = kDec,
    bool addLn = false ) [inherited]
```

Print an unsigned long integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.12.3.17 println() [1/10]

```
size_t Writer::println (
    char c ) [inline], [inherited]
```

Print a single character to the output stream, adding a new line character at the end.

- *c* is the character to output.

Returns

the number of bytes sent to the output stream.

9.12.3.18 println() [2/10]

```
size_t Writer::println (
    const char * str ) [inline], [inherited]
```

Print a null-terminated string to the output stream, adding a new line character at the end.

- `str` is the null-terminated string to output.

Returns

the number of bytes sent to the output stream.

9.12.3.19 println() [3/10]

```
size_t Writer::println (
    const uint8_t * buf,
    size_t size ) [inline], [inherited]
```

Print a number of bytes to the output stream, adding a new line character at the end.

- `buf` is the buffer containing bytes to output.
- `size` is the number of bytes from the buffer to output.

Returns

the number of bytes sent to the output stream.

9.12.3.20 println() [4/10]

```
size_t Writer::println (
    double d,
    int digits = 2 ) [inline], [inherited]
```

Print a floating point number to the output stream, adding a new line character at the end.

- `d` is the floating point number to output.
- `digits` is the number of decimal digits to output; the default is 2.

Returns

the number of bytes sent to the output stream.

9.12.3.21 `println()` [5/10]

```
size_t Writer::println (  
    int n,  
    int base = kDec ) [inline], [inherited]
```

Print an integer to the output stream, adding a new line character at the end.

- `n` is the integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.12.3.22 `println()` [6/10]

```
size_t Writer::println (  
    int8_t n,  
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit integer to the output stream, adding a new line character at the end.

- `n` is the integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.12.3.23 `println()` [7/10]

```
size_t Writer::println (  
    long n,  
    int base = kDec ) [inline], [inherited]
```

Print a long integer to the output stream, adding a new line character at the end.

- `n` is the long integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.12.3.24 println() [8/10]

```
size_t Writer::println (
    uint8_t n,
    int base = kDec ) [inline], [inherited]
```

Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.12.3.25 println() [9/10]

```
size_t Writer::println (
    unsigned int n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned integer to the output stream, adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.12.3.26 println() [10/10]

```
size_t Writer::println (
    unsigned long n,
    int base = kDec ) [inline], [inherited]
```

Print an unsigned long integer to the output stream, adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).

Returns

the number of bytes sent to the output stream.

9.12.3.27 read()

```
int Serial3::read ( ) [virtual]
```

Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).

Returns

the next byte, or -1 if there is nothing to read in the input stream before timeout expires.

Implements [Reader](#).

9.12.3.28 readBytes() [1/2]

```
size_t Reader::readBytes (
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, terminating if length characters have been read or the function times out. The result is *NOT* null-terminated.

- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout).

9.12.3.29 readBytes() [2/2]

```
size_t Reader::readBytes (
    uint8_t * buffer,
    size_t length ) [inline], [inherited]
```

Read bytes (uint8_t) from the input stream into a buffer, terminating if length bytes have been read or the function times out.

- `buffer` a pointer to where the bytes read will be stored.
- `length` the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout).

9.12.3.30 readBytesUntil() [1/2]

```
size_t Reader::readBytesUntil (
    char terminator,
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, terminating when the terminator character is encountered, or if length characters have been read, or if the function times out. The result is *NOT* null-terminated.

- `terminator` a character that when encountered causes the function to return.
- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.12.3.31 readBytesUntil() [2/2]

```
size_t Reader::readBytesUntil (
    uint8_t terminator,
    uint8_t * buffer,
    size_t length ) [inline], [inherited]
```

Read bytes (`uint8_t`) from the input stream into a buffer, terminating when the terminator byte is encountered, or if length bytes have been read, or if the function times out.

- `terminator` a byte that when encountered causes the function to return.
- `buffer` a pointer to where the bytes read will be stored.
- `length` the maximum number of bytes to read.

Returns

the number of bytes placed in the buffer (0 means no data were read prior to timeout or detecting the terminator character).

9.12.3.32 readFloat() [1/2]

```
bool Reader::readFloat (
    float * result ) [inherited]
```

Return the first valid float value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit.

- `result` is a pointer to where the float will be stored.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.12.3.33 readFloat() [2/2]

```
bool Reader::readFloat (
    float * result,
    char skipChar ) [inherited]
```

Return the first valid float value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the float is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the float will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid float is found prior to timeout; false otherwise.

9.12.3.34 readLine()

```
size_t Reader::readLine (
    char * buffer,
    size_t length ) [inherited]
```

Read characters from the input stream into a buffer, until it reaches EOL, or if length characters have been read, or if it times out. The result *IS* null-terminated.

- `buffer` a pointer to where the characters read will be stored.
- `length` the maximum number of characters to read.

Returns

the number of characters placed in the buffer (0 means no data were read prior to timeout or detecting EOL).

9.12.3.35 readLong() [1/2]

```
bool Reader::readLong (
    long * result ) [inherited]
```

Return the first valid long integer value from the stream.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit.

- `result` is a pointer to where the long integer will be stored.

Returns

true if a valid integer is found prior to timeout; false otherwise.

9.12.3.36 readLong() [2/2]

```
bool Reader::readLong (
    long * result,
    char skipChar ) [inherited]
```

Return the first valid long integer value from the stream, ignoring selected characters.

Initial characters that are not digits (or the minus sign) are skipped; the integer is terminated by the first character that is not a digit and is not one of the skip characters. This allows format characters (typically commas) to be ignored on input.

- `result` is a pointer to where the long integer will be stored.
- `skipChar` is a character that will be ignored on input.

Returns

true if a valid long integer is found prior to timeout; false otherwise.

9.12.3.37 setTimeout()

```
void Reader::setTimeout (
    unsigned long milliseconds ) [inline], [inherited]
```

Sets maximum milliseconds to wait for stream data, default is 1 second.

- `milliseconds` the length of the timeout period in milliseconds.

9.12.3.38 start()

```
void Serial3::start (
    unsigned long baudRate,
    UsartSerialConfiguration config = kSerial_8N1 ) [inline]
```

Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial3](#) on USART3.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).
- `config` sets the configuration in term of data bits, parity, and stop bits. If omitted, the default is 8 data bits, no parity, and 1 stop bit.

9.12.3.39 stop()

```
void Serial3::stop ( ) [inline]
```

Stops buffered serial communications using [Serial3](#) on USART3 by deconfiguring the hardware and turning off interrupts.

After calling this function, Arduino pins 0 and 1 are released and available for use as ordinary digital pins.

If you want to use [Serial3](#) again for communications, you must call [start\(\)](#) again.

9.12.3.40 write() [1/4]

```
size_t Serial3::write (
    char c ) [virtual]
```

Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).

- `c` the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.12.3.41 write() [2/4]

```
size_t Serial3::write (
    const char * buffer,
    size_t size ) [virtual]
```

Write a given number of characters from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const char* *buffer*, size_t *size* \)](#).

- *buffer* the buffer of characters to write.
- *size* the number of characters to write

Returns

the number of bytes written.

Implements [Writer](#).

9.12.3.42 write() [3/4]

```
size_t Serial3::write (
    const char * str ) [virtual]
```

Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char* *str* \)](#).

- *str* the string to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.12.3.43 write() [4/4]

```
size_t Serial3::write (
    const uint8_t * buffer,
    size_t size ) [virtual]
```

Write a given number of bytes from a buffer to the output stream. This implements the pure virtual function [Writer::write\(const uint8_t* *buffer*, size_t *size* \)](#).

- *buffer* the buffer of bytes to write.
- *size* the number of bytes to write

Returns

the number of bytes written.

Implements [Writer](#).

The documentation for this class was generated from the following files:

- [USART3.h](#)
- [USART3.cpp](#)

9.13 SPI::SPISettings Class Reference

A class that binds settings for configuring SPI transmissions.

```
#include <SPI.h>
```

Public Member Functions

- [SPISettings](#) (uint32_t maxSpeed, uint8_t bitOrder, uint8_t dataMode)
The constructor builds an [SPISettings](#) object out of three parameters describing the maximum transmission speed, the data order (most or least significant bit first), and the data mode (phase and polarity). Note that bit order extends to byte order when passing multibyte integers.
- [SPISettings](#) ()
The constructor builds an [SPISettings](#) object with default settings corresponding to a maximum transmission speed of 8 MHz, most significant bit first, and kSpiMode0.
- uint8_t [getSpcr](#) () const
Return the appropriate configure value for the SPCR register.
- uint8_t [getSpsr](#) () const
Return the appropriate configure value for the SPSR register.

9.13.1 Detailed Description

A class that binds settings for configuring SPI transmissions.

The [SPISettings](#) object is used to configure the SPI hardware. The three parameters are combined into a single [SPISettings](#) object, which is passed to [SPI::configure\(\)](#). You need to configure the SPI subsystem in this way before transmitting any data. The configuration remains in effect until explicitly changed by another call to [SPI::configure\(\)](#) or the SPI subsystem is disabled by a call to [SPI::disable\(\)](#).

This class is taken almost verbatim from the Arduino library [SPISettings](#) class created by Matthijs Kooijman and licensed under terms of either the GNU General Public License version 2 or the GNU Lesser General Public License version 2.1.

The implementation makes clever use of GCC intrinsic functions to do essentially all the heavy lifting at compile time whenever the SPI parameters are compile-time constants, producing very small and efficient code in this case. My modifications reformat the code to the AVRTools library conventions and adapt the interface to align with the AVRTools SPI implementation.

9.13.2 Constructor & Destructor Documentation

9.13.2.1 SPISettings()

```
SPI::SPISettings::SPISettings (
    uint32_t maxSpeed,
    uint8_t bitOrder,
    uint8_t dataMode ) [inline]
```

The constructor builds an [SPISettings](#) object out of three parameters describing the maximum transmission speed, the data order (most or least significant bit first), and the data mode (phase and polarity). Note that bit order extends to byte order when passing multibyte integers.

The code is designed to be exceptionally efficient and small if all three parameters are compile-time constants.

- `maxSpeed` the maximum speed of transmission, in herz (Hz). For a SPI chip rated up to 16 MHz, use 16000000.
- `bitOrder` whether least significant or most significant bit is first. Pass either `kMsbFirst` or `kLsbFirst`.
- `dataMode` sets the data mode (phase and polarity) for SPI communications. Pass one of `kSpiMode0`, `kSpiMode1`, `kSpiMode2`, or `kSpiMode3`.

9.13.3 Member Function Documentation

9.13.3.1 `getSpcr()`

```
uint8_t SPI::SPISettings::getSpcr ( ) const [inline]
```

Return the appropriate configure value for the SPCR register.

Returns

a value to load in the SPCR register to configure the SPI hardware.

9.13.3.2 `getSpsr()`

```
uint8_t SPI::SPISettings::getSpsr ( ) const [inline]
```

Return the appropriate configure value for the SPSR register.

Returns

a value to load in the SPSR register to configure the SPI hardware.

The documentation for this class was generated from the following file:

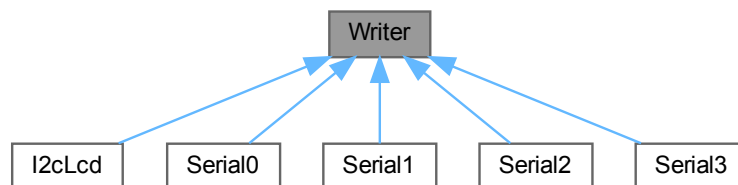
- [SPI.h](#)

9.14 Writer Class Reference

This is an abstract class defining a generic interface to write numbers and strings to a sequential stream of bytes (such as a serial output device).

```
#include <Writer.h>
```

Inheritance diagram for Writer:



Public Types

- enum `IntegerOutputBase` { `kBin` = 2 , `kOct` = 8 , `kDec` = 10 , `kHex` = 16 }

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Public Member Functions

- virtual `size_t write` (char c)=0
Pure virtual function that writes a single character to the output stream.
- virtual `size_t write` (const char *str)=0
Pure virtual function that writes a null-terminated string to the output stream.
- virtual `size_t write` (const char *buffer, `size_t` size)=0
Pure virtual function that writes a given number of characters from a buffer to the output stream.
- virtual `size_t write` (const `uint8_t` *buffer, `size_t` size)=0
Pure virtual function that writes a given number of bytes from a buffer to the output stream.
- virtual void `flush` ()=0
Pure virtual function to flush the output stream. When this function returns, all previously written data will have been transmitted through the underlying output stream.
- `size_t print` (const char *str, bool addLn=false)
Print a null-terminated string to the output stream, with or without adding a new line character at the end.
- `size_t print` (const `uint8_t` *buf, `size_t` size, bool addLn=false)
Print a number of bytes to the output stream, with or without adding a new line character at the end.
- `size_t print` (char c, bool addLn=false)
Print a single character to the output stream, with or without adding a new line character at the end.
- `size_t print` (`int8_t` n, int base=`kDec`, bool addLn=false)
Print an 8-bit integer to the output stream, with or without adding a new line character at the end.
- `size_t print` (`uint8_t` n, int base=`kDec`, bool addLn=false)
Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.
- `size_t print` (int n, int base=`kDec`, bool addLn=false)
Print an integer to the output stream, with or without adding a new line character at the end.
- `size_t print` (unsigned int n, int base=`kDec`, bool addLn=false)
Print an unsigned integer to the output stream, with or without adding a new line character at the end.
- `size_t print` (long n, int base=`kDec`, bool addLn=false)
Print a long integer to the output stream, with or without adding a new line character at the end.
- `size_t print` (unsigned long n, int base=`kDec`, bool addLn=false)
Print an unsigned long integer to the output stream, with or without adding a new line character at the end.
- `size_t print` (double d, int digits=2, bool addLn=false)
Print a floating point number to the output stream, with or without adding a new line character at the end.
- `size_t println` (const char *str)
Print a null-terminated string to the output stream, adding a new line character at the end.
- `size_t println` (const `uint8_t` *buf, `size_t` size)
Print a number of bytes to the output stream, adding a new line character at the end.
- `size_t println` (char c)
Print a single character to the output stream, adding a new line character at the end.
- `size_t println` (`int8_t` n, int base=`kDec`)
Print an 8-bit integer to the output stream, adding a new line character at the end.

- `size_t println (uint8_t n, int base=kDec)`
Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (int n, int base=kDec)`
Print an integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned int n, int base=kDec)`
Print an unsigned integer to the output stream, adding a new line character at the end.
- `size_t println (long n, int base=kDec)`
Print a long integer to the output stream, adding a new line character at the end.
- `size_t println (unsigned long n, int base=kDec)`
Print an unsigned long integer to the output stream, adding a new line character at the end.
- `size_t println (double d, int digits=2)`
Print a floating point number to the output stream, adding a new line character at the end.
- `size_t println ()`
Print a new line to the output stream.

9.14.1 Detailed Description

This is an abstract class defining a generic interface to write numbers and strings to a sequential stream of bytes (such as a serial output device).

It implements functions to convert various integers and floating point numbers into a sequence of bytes (so it is not a pure interface class). These functions depend on a small set of lower-level functions that are purely abstract and must be implemented by classes deriving from [Writer](#).

[Serial0](#) is an example of a class that derives from [Writer](#) by implementing the purely abstract functions in [Writer](#).

9.14.2 Member Enumeration Documentation

9.14.2.1 IntegerOutputBase

enum [Writer::IntegerOutputBase](#)

An enumeration that defines the number that will be used as the base for representing integer quantities as a string of characters.

Enumerator

kBin	Produce a binary representation of integers (e.g., 11 is output as 0b1011)
kOct	Produce an octal representation of integers (e.g, 11 is output as 013)
kDec	Produce a decimal representation of integers (e.g., 11 is output as 11.
kHex	Produce a hexadecimal representation of integers (e.g., 11 is output as 0x0b)

9.14.3 Member Function Documentation

9.14.3.1 `print()` [1/10]

```
size_t Writer::print (
    char c,
    bool addLn = false )
```

Print a single character to the output stream, with or without adding a new line character at the end.

- `c` is the character to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.2 `print()` [2/10]

```
size_t Writer::print (
    const char * str,
    bool addLn = false )
```

Print a null-terminated string to the output stream, with or without adding a new line character at the end.

- `str` is the null-terminated string to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.3 `print()` [3/10]

```
size_t Writer::print (
    const uint8_t * buf,
    size_t size,
    bool addLn = false )
```

Print a number of bytes to the output stream, with or without adding a new line character at the end.

- `buf` is the buffer containing bytes to output.
- `size` is the number of bytes from the buffer to output.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.4 print() [4/10]

```
size_t Writer::print (
    double d,
    int digits = 2,
    bool addLn = false )
```

Print a floating point number to the output stream, with or without adding a new line character at the end.

- `d` is the floating point number to output.
- `digits` is the number of decimal digits to output; the default is 2.
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.5 print() [5/10]

```
size_t Writer::print (
    int n,
    int base = kDec,
    bool addLn = false )
```

Print an integer to the output stream, with or without adding a new line character at the end.

- `n` is the integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- `addLn` if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.6 print() [6/10]

```
size_t Writer::print (
    int8_t n,
    int base = kDec,
    bool addLn = false )
```

Print an 8-bit integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.7 print() [7/10]

```
size_t Writer::print (
    long n,
    int base = kDec,
    bool addLn = false )
```

Print a long integer to the output stream, with or without adding a new line character at the end.

- *n* is the long integer to output.
- *base* is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.8 print() [8/10]

```
size_t Writer::print (
    uint8_t n,
    int base = kDec,
    bool addLn = false ) [inline]
```

Print an 8-bit unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the integer to output.
- *base* is the base used to represent the number; should be one of *IntegerOutputBase*; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.9 print() [9/10]

```
size_t Writer::print (
    unsigned int n,
    int base = kDec,
    bool addLn = false ) [inline]
```

Print an unsigned integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned integer to output.
- *base* is the base used to represent the number; should be one of *IntegerOutputBase*; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.10 print() [10/10]

```
size_t Writer::print (
    unsigned long n,
    int base = kDec,
    bool addLn = false )
```

Print an unsigned long integer to the output stream, with or without adding a new line character at the end.

- *n* is the unsigned long integer to output.
- *base* is the base used to represent the number; should be one of IntegerOutputBase; defaults to decimal representation (*kDec*).
- *addLn* if true, a new line character is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.14.3.11 println() [1/10]

```
size_t Writer::println (
    char c ) [inline]
```

Print a single character to the output stream, adding a new line character at the end.

- *c* is the character to output.

Returns

the number of bytes sent to the output stream.

9.14.3.12 println() [2/10]

```
size_t Writer::println (
    const char * str ) [inline]
```

Print a null-terminated string to the output stream, adding a new line character at the end.

- *str* is the null-terminated string to output.

Returns

the number of bytes sent to the output stream.

9.14.3.13 println() [3/10]

```
size_t Writer::println (
    const uint8_t * buf,
    size_t size ) [inline]
```

Print a number of bytes to the output stream, adding a new line character at the end.

- `buf` is the buffer containing bytes to output.
- `size` is the number of bytes from the buffer to output.

Returns

the number of bytes sent to the output stream.

9.14.3.14 println() [4/10]

```
size_t Writer::println (
    double d,
    int digits = 2 ) [inline]
```

Print a floating point number to the output stream, adding a new line character at the end.

- `d` is the floating point number to output.
- `digits` is the number of decimal digits to output; the default is 2.

Returns

the number of bytes sent to the output stream.

9.14.3.15 println() [5/10]

```
size_t Writer::println (
    int n,
    int base = kDec ) [inline]
```

Print an integer to the output stream, adding a new line character at the end.

- `n` is the integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.14.3.16 `println()` [6/10]

```
size_t Writer::println (
    int8_t n,
    int base = kDec ) [inline]
```

Print an 8-bit integer to the output stream, adding a new line character at the end.

- `n` is the integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.14.3.17 `println()` [7/10]

```
size_t Writer::println (
    long n,
    int base = kDec ) [inline]
```

Print a long integer to the output stream, adding a new line character at the end.

- `n` is the long integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.14.3.18 `println()` [8/10]

```
size_t Writer::println (
    uint8_t n,
    int base = kDec ) [inline]
```

Print an 8-bit unsigned integer to the output stream, adding a new line character at the end.

- `n` is the unsigned integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.14.3.19 println() [9/10]

```
size_t Writer::println (
    unsigned int n,
    int base = kDec ) [inline]
```

Print an unsigned integer to the output stream, adding a new line character at the end.

- `n` is the unsigned integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.14.3.20 println() [10/10]

```
size_t Writer::println (
    unsigned long n,
    int base = kDec ) [inline]
```

Print an unsigned long integer to the output stream, adding a new line character at the end.

- `n` is the unsigned long integer to output.
- `base` is the base used to represent the number; should be one of `IntegerOutputBase`; defaults to decimal representation (`kDec`).

Returns

the number of bytes sent to the output stream.

9.14.3.21 write() [1/4]

```
size_t Writer::write (
    char c ) [pure virtual]
```

Pure virtual function that writes a single character to the output stream.

- `c` the character to be written.

Returns

the number of bytes written.

Implemented in [I2cLcd](#), [Serial0](#), [Serial1](#), [Serial2](#), and [Serial3](#).

9.14.3.22 write() [2/4]

```
size_t Writer::write (
    const char * buffer,
    size_t size ) [pure virtual]
```

Pure virtual function that writes a given number of characters from a buffer to the output stream.

- *buffer* the buffer of characters to write.
- *size* the number of characters to write

Returns

the number of bytes written.

Implemented in [I2cLcd](#), [Serial0](#), [Serial1](#), [Serial2](#), and [Serial3](#).

9.14.3.23 write() [3/4]

```
size_t Writer::write (
    const char * str ) [pure virtual]
```

Pure virtual function that writes a null-terminated string to the output stream.

- *str* the string to be written.

Returns

the number of bytes written.

Implemented in [I2cLcd](#), [Serial0](#), [Serial1](#), [Serial2](#), and [Serial3](#).

9.14.3.24 write() [4/4]

```
size_t Writer::write (
    const uint8_t * buffer,
    size_t size ) [pure virtual]
```

Pure virtual function that writes a given number of bytes from a buffer to the output stream.

- *buffer* the buffer of bytes to write.
- *size* the number of bytes to write

Returns

the number of bytes written.

Implemented in [I2cLcd](#), [Serial0](#), [Serial1](#), [Serial2](#), and [Serial3](#).

The documentation for this class was generated from the following files:

- [Writer.h](#)
- [Writer.cpp](#)

Chapter 10

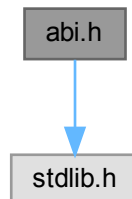
File Documentation

10.1 abi.h File Reference

This file provides certain functions needed to complete the avr-gcc C++ ABI. You never need to include this file, and you only need to link against abi.cpp if you encounter certain link errors.

```
#include <stdlib.h>
```

Include dependency graph for abi.h:



10.1.1 Detailed Description

This file provides certain functions needed to complete the avr-gcc C++ ABI. You never need to include this file, and you only need to link against abi.cpp if you encounter certain link errors.

If when building your project you get link-time errors about undefined references to symbols of the form `__cxa_XXX` (e.g., `__cxa_pure_virtual`), then you should link your project against abi.cpp (there is no need to include [abi.h](#) in any of your sources).

If you don't encounter such errors, you can completely disregard both [abi.h](#) and `abi.cpp`.

10.2 abi.h

[Go to the documentation of this file.](#)

```

00001 /*
00002     abi.h - C++ ABI support missing from avr-gcc.
00003     This is part of the AVRTools library.
00004     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00005
00006     This program is free software: you can redistribute it and/or modify
00007     it under the terms of the GNU General Public License as published by
00008     the Free Software Foundation, either version 3 of the License, or
00009     (at your option) any later version.
00010
00011     This program is distributed in the hope that it will be useful,
00012     but WITHOUT ANY WARRANTY; without even the implied warranty of
00013     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00014     GNU General Public License for more details.
00015
00016     You should have received a copy of the GNU General Public License
00017     along with this program. If not, see <http://www.gnu.org/licenses/>.
00018 */
00019
00020
00021
00039 #ifndef abi_h
00040 #define abi_h
00041
00042
00043 #include <stdlib.h>
00044
00045 __extension__ typedef int __guard __attribute__((mode (__DI__)));
00046
00047 extern "C"
00048 {
00049     int __cxa_guard_acquire( __guard* );
00050     void __cxa_guard_release ( __guard* );
00051     void __cxa_guard_abort ( __guard* );
00052
00053     void __cxa_pure_virtual() __attribute__((__noreturn__));
00054     void __cxa_deleted_virtual() __attribute__((__noreturn__));
00055 }
00056
00057
00058
00059
00060
00061 #endif

```

10.3 Analog2Digital.h File Reference

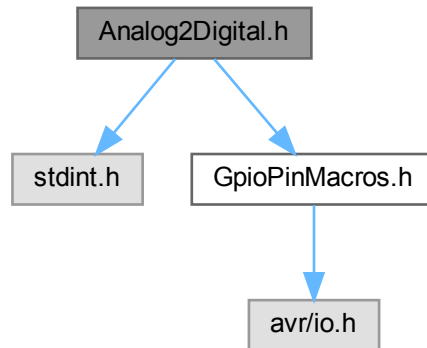
This file provides functions that access the analog-to-digital conversion capability of the ATmega328 and ATmega2560 microcontrollers.

```

#include <stdint.h>
#include "GpioPinMacros.h"

```


Include dependency graph for Analog2Digital.h:



Macros

- #define `readGpioPinAnalog`(pinName)
Read the analog value of the pin.

Enumerations

- enum `A2DVoltageReference` { `kA2dReferenceAREF` , `kA2dReferenceAVCC` , `kA2dReference11V` , `kA2dReference256V` }
Constants representing voltage references.

Functions

- int `readA2D` (int8_t channel)
Read an analog voltage value.
- uint16_t `readGpioPinAnalogV` (const `GpioPinVariable` &pinVar)
Read the analog value of the pin.
- void `initA2D` (uint8_t ref=`kA2dReferenceAVCC`)
Initialize the analog-to-digital system.
- void `turnOffA2D` ()
Turn off the analog-to-digital system.
- void `setA2DVoltageReference` (`A2DVoltageReference` ref)
Set the voltage reference for the analog-to-digital system.
- void `setA2DVoltageReferenceAREF` ()
Set the voltage reference for the analog-to-digital system to AREF.
- void `setA2DVoltageReferenceAVCC` ()
Set the voltage reference for the analog-to-digital system to AREF.
- void `setA2DVoltageReference11V` ()
Set the voltage reference for the analog-to-digital system to AREF.
- void `setA2DVoltageReference256V` ()
Set the voltage reference for the analog-to-digital system to AREF.

10.3.1 Detailed Description

This file provides functions that access the analog-to-digital conversion capability of the ATmega328 and ATmega2560 microcontrollers.

To use these functions, include [Analog2Digital.h](#) in your source code and link against `Analog2Digital.cpp`.

10.3.2 Macro Definition Documentation

10.3.2.1 readGpioPinAnalog

```
#define readGpioPinAnalog(  
    pinName )
```

Read the analog value of the pin.

This function returns a number between 0 and 1023 that corresponds to voltage between 0 and a maximum reference value. The reference value is set using one of the `setA2DVoltageReferenceXXX()` functions.

- `pinName` a pin name macro generated by [GpioPinAnalog\(\)](#).

Returns

an value between 0 and 1023.

Note

Before calling this function must fist initialize the analog-to-digital sub-system by calling [initA2D\(\)](#).

10.3.3 Enumeration Type Documentation

10.3.3.1 A2DVoltageReference

```
enum A2DVoltageReference
```

Constants representing voltage references.

Enumerator

<code>kA2dReferenceAREF</code>	Reference is AREF pin, internal VREF turned off.
<code>kA2dReferenceAVCC</code>	Reference is AVCC pin, internal VREF turned off.
<code>kA2dReference11V</code>	Reference is internal 1.1V VREF.
<code>kA2dReference256V</code>	Reference is internal 2.56V VREF (only available on ATmega2560)

10.3.4 Function Documentation

10.3.4.1 initA2D()

```
void initA2D (
    uint8_t ref = kA2dReferenceAVCC )
```

Initialize the analog-to-digital system.

You must call this function before using any of the analog-to-digital functions.

- `ref` provides the voltage reference to be used for analog-to-digital conversions. Pass one of the constants from enum `A2DVoltageReference`. If no value is provided, the default is `kA2dReferenceAVCC`.

Note

This function only works for CPU clocks running at either 8 MHz, 12 MHz, or 16 MHz.

10.3.4.2 readA2D()

```
int readA2D (
    int8_t channel )
```

Read an analog voltage value.

Voltage is read relative to the currently set reference value.

- `channel` is an ADC channel number (between 0 and 7 on ATmega328; between 0 and 15 on ATmega2560).

Returns

a number between 0 and 1023.

Note

Generally users will not call this function but instead call `readPinAnalog()` passing it a pin name macro generated by `Analog()`.

10.3.4.3 readGpioPinAnalogV()

```
uint16_t readGpioPinAnalogV (
    const GpioPinVariable & pinVar ) [inline]
```

Read the analog value of the pin.

This function returns a number between 0 and 1023 that corresponds to voltage between 0 and a maximum reference value. The reference value is set using one of the `setA2DVoltageReferenceXXX()` functions.

- `pinVar` a pin variable that has analog-to-digital capabilities (i.e., initialized with `makeGpioVarFromGpioPinAnalog()`).

Returns

an value between 0 and 1023.

Note

Before calling this function must fist initialize the analog-to-digital sub-system by calling `initA2D()`.

10.3.4.4 setA2DVoltageReference()

```
void setA2DVoltageReference (
    A2DVoltageReference ref )
```

Set the voltage reference for the analog-to-digital system.

After your have initialized the analog-to-digital system with `initA2D()`, you can use this function to change the voltage reference.

- `ref` provides the voltage reference to be used for analog-to-digital conversions. Pass one of the constants from `enum A2DVoltageReference`.

10.3.4.5 setA2DVoltageReference11V()

```
void setA2DVoltageReference11V ( ) [inline]
```

Set the voltage reference for the analog-to-digital system to AREF.

This is an inline synonym for `setA2DVoltageReference(kA2dReference11V)`

10.3.4.6 setA2DVoltageReference256V()

```
void setA2DVoltageReference256V ( ) [inline]
```

Set the voltage reference for the analog-to-digital system to AREF.

This is an inline synonym for setA2DVoltageReference(kA2dReference256V)

Note

this function is only available on ATmega2560 (not on the ATmega328).

10.3.4.7 setA2DVoltageReferenceAREF()

```
void setA2DVoltageReferenceAREF ( ) [inline]
```

Set the voltage reference for the analog-to-digital system to AREF.

This is an inline synonym for setA2DVoltageReference(kA2dReferenceAREF)

10.3.4.8 setA2DVoltageReferenceAVCC()

```
void setA2DVoltageReferenceAVCC ( ) [inline]
```

Set the voltage reference for the analog-to-digital system to AREF.

This is an inline synonym for setA2DVoltageReference(kA2dReferenceAVCC)

10.4 Analog2Digital.h

[Go to the documentation of this file.](#)

```
00001 /*
00002     Analog2Digital.h - A library for analog-to-digital conversions.
00003     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007     This program is free software: you can redistribute it and/or modify
00008     it under the terms of the GNU General Public License as published by
00009     the Free Software Foundation, either version 3 of the License, or
00010     (at your option) any later version.
00011
00012     This program is distributed in the hope that it will be useful,
00013     but WITHOUT ANY WARRANTY; without even the implied warranty of
00014     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015     GNU General Public License for more details.
00016
00017     You should have received a copy of the GNU General Public License
00018     along with this program. If not, see <http://www.gnu.org/licenses/>.
00019 */
00020
00021
00033 #ifndef Analog2Digital_h
00034 #define Analog2Digital_h
00035
```

```

00036 #include <stdint.h>
00037
00038 #include "GpioPinMacros.h"
00039
00040
00041 #if defined(__AVR_ATmega2560__)
00042
00043
00048 enum A2DVoltageReference
00049 {
00050     kA2dReferenceAREF = 0x00,
00051     kA2dReferenceAVCC = 0x01,
00052     kA2dReference11V  = 0x02,
00053     kA2dReference256V = 0x03
00054 };
00055
00056 #else
00057
00058 enum A2DVoltageReference
00059 {
00060     kA2dReferenceAREF = 0x00,    // 0x00 -> AREF pin, internal VREF turned off
00061     kA2dReferenceAVCC = 0x01,    // 0x01 -> AVCC pin, internal VREF turned off
00062     kA2dReference11V  = 0x03    // 0x03 -> Internal 1.1V VREF
00063 };
00064
00065 #endif
00066
00067 /*
00068     The following macro is not intended for end-user use; it is needed to support the pin naming
00069     macros in conjunction with the C/C++ preprocessor's re-scanning rules.
00071 */
00072
00073 #define _readGpioPinAnalog( ddr, port, pin, nbr, adc, ocr, com, tccr )    readA2D( adc )
00074
00075
00076
00077
00095 #define readGpioPinAnalog( pinName )    _readGpioPinAnalog( pinName )
00096
00097
00098 int readA2D( int8_t channel );
00099
00100
00101
00117 inline uint16_t readGpioPinAnalogV( const GpioPinVariable& pinVar )
00118 {
00119     return readA2D( pinVar.adcNbr() );
00120 }
00121
00122
00135 void initA2D( uint8_t ref = kA2dReferenceAVCC );
00136
00137
00138
00144 void turnOffA2D();
00145
00146
00157 void setA2DVoltageReference( A2DVoltageReference ref );
00158
00159
00160
00167 inline void setA2DVoltageReferenceAREF()
00168 { setA2DVoltageReference( kA2dReferenceAREF ); }
00169
00170
00171
00178 inline void setA2DVoltageReferenceAVCC()
00179 { setA2DVoltageReference( kA2dReferenceAVCC ); }
00180
00181
00182
00189 inline void setA2DVoltageReference11V()
00190 { setA2DVoltageReference( kA2dReference11V ); }
00191
00192
00193 #if defined(__AVR_ATmega2560__)
00194
00195
00204 inline void setA2DVoltageReference256V()
00205 { setA2DVoltageReference( kA2dReference256V ); }

```

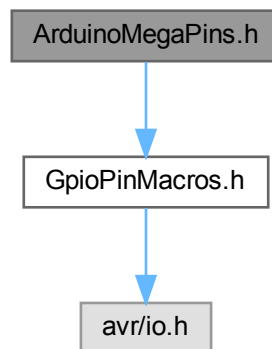
```
00206
00207 #endif
00208
00209
00210
00226 int readA2D( int8_t channel );
00227
00228 #endif
```

10.5 ArduinoMegaPins.h File Reference

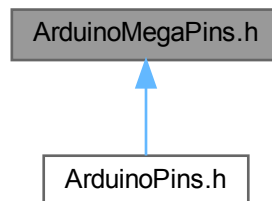
This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file [ArduinoPins.h](#), which in turn includes this file (when compiling for Arduino Uno targets).

```
#include "GpioPinMacros.h"
```

Include dependency graph for ArduinoMegaPins.h:



This graph shows which files directly or indirectly include this file:



10.5.1 Detailed Description

This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file [ArduinoPins.h](#), which in turn includes this file (when compiling for Arduino Uno targets).

The standard Arduino Uno digital pins will be defined as pPin00 through pPin53.

The standard Arduino Uno analog pins will be defined as pPinA00 through pPinA15.

Additionally, the I2C SDA and SCL pins are also defined as pSDA and pSCL (these are synonyms for pPin20 and pPin21, respectively).

10.6 ArduinoMegaPins.h

[Go to the documentation of this file.](#)

```
00001 /*
00002     ArduinoMegaPins.h - Macros naming the pins on the Arduino Mega.
00003     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007     This program is free software: you can redistribute it and/or modify
00008     it under the terms of the GNU General Public License as published by
00009     the Free Software Foundation, either version 3 of the License, or
00010     (at your option) any later version.
00011
00012     This program is distributed in the hope that it will be useful,
00013     but WITHOUT ANY WARRANTY; without even the implied warranty of
00014     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015     GNU General Public License for more details.
00016
00017     You should have received a copy of the GNU General Public License
00018     along with this program. If not, see <http://www.gnu.org/licenses/>.
00019 */
00020
00021
00040 #ifndef ArduinoMegaPins_h
00041 #define ArduinoMegaPins_h
00042
00043 #ifndef ArduinoPinsDefined
00044 #define ArduinoPinsDefined
00045 #else
00046 #error "Only include one Arduino model pin definition file; more than one appears to be included"
00047 #endif
00048
00049
00050
00051 #include "GpioPinMacros.h"
00052
00053
00054 #define pPinA00          GpioPinAnalog( F, 0, 0 )           // PF0, ADC0
00055 #define pPinA01          GpioPinAnalog( F, 1, 1 )           // PF1, ADC1
00056 #define pPinA02          GpioPinAnalog( F, 2, 2 )           // PF2, ADC2
00057 #define pPinA03          GpioPinAnalog( F, 3, 3 )           // PF3, ADC3
00058 #define pPinA04          GpioPinAnalog( F, 4, 4 )           // PF4, ADC4, TCK
00059 #define pPinA05          GpioPinAnalog( F, 5, 5 )           // PF5, ADC5, TMS
00060 #define pPinA06          GpioPinAnalog( F, 6, 6 )           // PF5, ADC6, TDO
00061 #define pPinA07          GpioPinAnalog( F, 7, 7 )           // PF5, ADC7, TDI
00062
00063 #define pPinA08          GpioPinAnalog( K, 0, 8 )           // PF0, ADC8, PCINT16
00064 #define pPinA09          GpioPinAnalog( K, 1, 9 )           // PF1, ADC9, PCINT17
00065 #define pPinA10          GpioPinAnalog( K, 2, 10 )          // PF2, ADC10, PCINT18
00066 #define pPinA11          GpioPinAnalog( K, 3, 11 )          // PF3, ADC11, PCINT19
00067 #define pPinA12          GpioPinAnalog( K, 4, 12 )          // PF4, ADC12, PCINT20
00068 #define pPinA13          GpioPinAnalog( K, 5, 13 )          // PF5, ADC13, PCINT21
00069 #define pPinA14          GpioPinAnalog( K, 6, 14 )          // PF5, ADC14, PCINT22
00070 #define pPinA15          GpioPinAnalog( K, 7, 15 )          // PF5, ADC15, PCINT23
00071
```



```

00072 #define pPin00      GpioPin( E, 0 )           // PE0, RXD0, PCINT8
00073 #define pPin01      GpioPin( E, 1 )           // PE1, TXD0, PCINT3
00074 #define pPin02      GpioPinPwm( E, 4, 3, B )   // PE4, INT4, OC3B
00075 #define pPin03      GpioPinPwm( E, 5, 3, C )   // PE5, INT5, OC3C
00076 #define pPin04      GpioPinPwm( G, 5, 0, B )   // PG5, OC0B
00077 #define pPin05      GpioPinPwm( E, 3, 3, A )   // PE3, AIN1, OC3A
00078
00079 #define pPin06      GpioPinPwm( H, 3, 4, A )   // PH3, OC4A, PCINT8
00080 #define pPin07      GpioPinPwm( H, 4, 4, B )   // PH4, OC4B
00081 #define pPin08      GpioPinPwm( H, 5, 4, C )   // PH5, OC4C
00082 #define pPin09      GpioPinPwm( H, 6, 2, B )   // PH6, OC2B
00083 #define pPin10      GpioPinPwm( B, 4, 2, A )   // PB4, OC2A, PCINT4
00084
00085 #define pPin11      GpioPinPwm( B, 5, 1, A )   // PB5, OC1A, PCINT5
00086 #define pPin12      GpioPinPwm( B, 6, 1, B )   // PB6, OC1B, PCINT6
00087 #define pPin13      GpioPinPwm( B, 7, 0, A )   // PB7, OC0A, PCINT7
00088 #define pPin14      GpioPin( J, 1 )           // PJ1, TXD3, PCINT10
00089 #define pPin15      GpioPin( J, 0 )           // PJ0, RXD3, PCINT9
00090
00091 #define pPin16      GpioPin( H, 1 )           // PH1, TXD2
00092 #define pPin17      GpioPin( H, 0 )           // PH0, RXD2
00093 #define pPin18      GpioPin( D, 3 )           // PD3, INT3, TXD1
00094 #define pPin19      GpioPin( D, 2 )           // PD2, INT2, RXD1
00095 #define pPin20      GpioPin( D, 1 )           // PD1, INT1, SDA
00096
00097 #define pPin21      GpioPin( D, 0 )           // PD0, INT0, SCL
00098 #define pPin22      GpioPin( A, 0 )           // PA0, AD0
00099 #define pPin23      GpioPin( A, 1 )           // PA1, AD1
00100 #define pPin24      GpioPin( A, 2 )           // PA2, AD2
00101 #define pPin25      GpioPin( A, 3 )           // PA3, AD3
00102
00103 #define pPin26      GpioPin( A, 4 )           // PA4, AD4
00104 #define pPin27      GpioPin( A, 5 )           // PA5, AD5
00105 #define pPin28      GpioPin( A, 6 )           // PA6, AD6
00106 #define pPin29      GpioPin( A, 7 )           // PA7, AD7
00107 #define pPin30      GpioPin( C, 7 )           // PC7, A15
00108
00109 #define pPin31      GpioPin( C, 6 )           // PC6, A14
00110 #define pPin32      GpioPin( C, 5 )           // PC5, A13
00111 #define pPin33      GpioPin( C, 4 )           // PC4, A12
00112 #define pPin34      GpioPin( C, 3 )           // PC3, A11
00113 #define pPin35      GpioPin( C, 2 )           // PC2, A10
00114
00115 #define pPin36      GpioPin( C, 1 )           // PC1, A9
00116 #define pPin37      GpioPin( C, 0 )           // PC0, A8
00117 #define pPin38      GpioPin( D, 7 )           // PD7, T0
00118 #define pPin39      GpioPin( G, 2 )           // PG2, ALE
00119 #define pPin40      GpioPin( G, 1 )           // PG1, RD
00120
00121 #define pPin41      GpioPin( G, 0 )           // PG0, WR
00122 #define pPin42      GpioPin( L, 7 )           // PL7
00123 #define pPin43      GpioPin( L, 6 )           // PL6
00124 #define pPin44      GpioPinPwm( L, 5, 5, C )   // PL5, OC5C
00125 #define pPin45      GpioPinPwm( L, 4, 5, B )   // PL4, OC5B
00126
00127 #define pPin46      GpioPinPwm( L, 3, 5, A )   // PL3, OC5A
00128 #define pPin47      GpioPin( L, 2 )           // PL2, T5
00129 #define pPin48      GpioPin( L, 1 )           // PL1, ICP5
00130 #define pPin49      GpioPin( L, 0 )           // PL0, ICP4
00131 #define pPin50      GpioPin( B, 3 )           // PB3, MISO, PCINT3
00132
00133 #define pPin51      GpioPin( B, 2 )           // PB2, MOSI, PCINT2
00134 #define pPin52      GpioPin( B, 1 )           // PB1, SCK, PCINT1
00135 #define pPin53      GpioPin( B, 0 )           // PB0, SS, PCINT0
00136
00137 #define pSDA        pPin20                   // I2C SDA
00138 #define pSCL        pPin21                   // I2C SCL
00139
00140 #define pSS         pPin53                   // SPI SS
00141 #define pMOSI       pPin51                   // SPI MOSI
00142 #define pMISO       pPin50                   // SPI MISO
00143 #define pSCK        pPin52                   // SPI SCK
00144
00145
00146 #endif

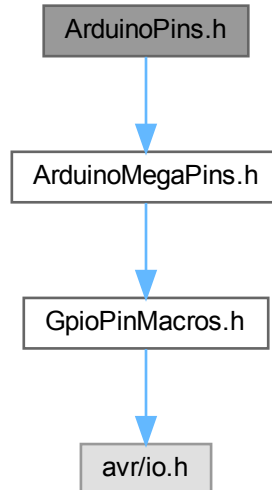
```

10.7 ArduinoPins.h File Reference

This file is the primary one that users should include to access and use the pin name macros.

```
#include "ArduinoMegaPins.h"
```

Include dependency graph for ArduinoPins.h:



10.7.1 Detailed Description

This file is the primary one that users should include to access and use the pin name macros.

Including this file will automatically include either the default Arduino Uno pin names (by including [ArduinoUnoPins.h](#)) or the default Arduino Mega pin names (by including [ArduinoMegaPins.h](#)).

The standard Arduino digital pins will be defined in the form `pPinNN` (where `NN` = 00 through 13 for Arduino Uno, and 00 through 53 for Arduino Mega).

The standard Arduino analog pins will be defined in the form `pPinAxx` (where `xx` = 00 through 07 for Arduino Uno, and `xx` = 00 through 15 for Arduino Mega).

10.8 ArduinoPins.h

[Go to the documentation of this file.](#)

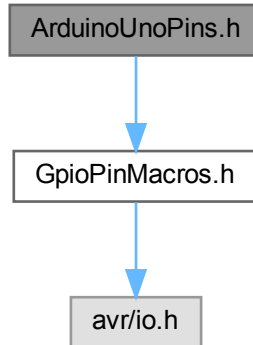
```
00001 /*
00002     ArduinoPins.h - Macros naming the Arduino pins (selects the appropriate variant).
00003     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007     This program is free software: you can redistribute it and/or modify
00008     it under the terms of the GNU General Public License as published by
00009     the Free Software Foundation, either version 3 of the License, or
00010     (at your option) any later version.
00011
00012     This program is distributed in the hope that it will be useful,
00013     but WITHOUT ANY WARRANTY; without even the implied warranty of
00014     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015     GNU General Public License for more details.
00016
00017     You should have received a copy of the GNU General Public License
00018     along with this program. If not, see <http://www.gnu.org/licenses/>.
00019 */
00020
00021
00022 #ifndef ArduinoPins_h
00023 #define ArduinoPins_h
00024
00025 #if defined(__AVR_ATmega328P__)
00026 #include "ArduinoUnoPins.h"
00027
00028 #elif defined(__AVR_ATmega2560__)
00029 #include "ArduinoMegaPins.h"
00030
00031 #else
00032 #error "Undefined AVR processor type"
00033 #endif
00034 #endif
```

10.9 ArduinoUnoPins.h File Reference

This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file [ArduinoPins.h](#), which in turn includes this file (when compiling for Arduino Uno targets).

```
#include "GpioPinMacros.h"
```

Include dependency graph for ArduinoUnoPins.h:



10.9.1 Detailed Description

This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file [ArduinoPins.h](#), which in turn includes this file (when compiling for Arduino Uno targets).

The standard Arduino Uno digital pins will be defined as pPin00 through pPin13.

The standard Arduino Uno analog pins will be defined as pPinA00 through pPinA07.

Additionally, the I2C SDA and SCL pins are also defined as pSDA and pSCL (these are synonyms for pPinA04 and pPinA05, respectively).

10.10 ArduinoUnoPins.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      ArduinoUnoPins.h - Macros naming the pins on the Arduino Uno.
00003      For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004      This is part of the AVRTools library.
00005      Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007      This program is free software: you can redistribute it and/or modify
00008      it under the terms of the GNU General Public License as published by
00009      the Free Software Foundation, either version 3 of the License, or
00010      (at your option) any later version.
00011
00012      This program is distributed in the hope that it will be useful,
00013      but WITHOUT ANY WARRANTY; without even the implied warranty of
00014      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015      GNU General Public License for more details.
00016
00017      You should have received a copy of the GNU General Public License
  
```

```

00018     along with this program.  If not, see <http://www.gnu.org/licenses/>.
00019 */
00020
00021
00041 #ifndef ArduinoUnoPins_h
00042 #define ArduinoUnoPins_h
00043
00044 #ifndef ArduinoPinsDefined
00045 #define ArduinoPinsDefined
00046 #else
00047 #error "Only include one Arduino model pin definition file; more than one appears to be included"
00048 #endif
00049
00050
00051
00052 #include "GpioPinMacros.h"
00053
00054
00055 #define pPinA00      GpioPinAnalog( C, 0, 0 )      // PC0, ADC0, PCINT8
00056 #define pPinA01      GpioPinAnalog( C, 1, 1 )      // PC1, ADC1, PCINT9
00057 #define pPinA02      GpioPinAnalog( C, 2, 2 )      // PC2, ADC2, PCINT10
00058 #define pPinA03      GpioPinAnalog( C, 3, 3 )      // PC3, ADC3, PCINT11
00059 #define pPinA04      GpioPinAnalog( C, 4, 4 )      // PC4, ADC4, SDA, PCINT12
00060 #define pPinA05      GpioPinAnalog( C, 5, 5 )      // PC5, ADC5, SCL, PCINT13
00061
00062 #define pPin00      GpioPin( D, 0 )      // PD0, RXD, PCINT16
00063 #define pPin01      GpioPin( D, 1 )      // PD1, TXD, PCINT17
00064 #define pPin02      GpioPin( D, 2 )      // PD2, INT0, PCINT18
00065 #define pPin03      GpioPinPwm( D, 3, 2, B )      // PD3, INT1, OC2B, PCINT19
00066 #define pPin04      GpioPin( D, 4 )      // PD4, T0, XCK, PCINT20
00067 #define pPin05      GpioPinPwm( D, 5, 0, B )      // PD5, T1, OC0B, PCINT21
00068 #define pPin06      GpioPinPwm( D, 6, 0, A )      // PD6, AIN0, OC0A, PCINT22
00069 #define pPin07      GpioPin( D, 7 )      // PD7, AIN1, PCINT23
00070 #define pPin08      GpioPin( B, 0 )      // PB0, ICPL, CLKO, PCINT0
00071 #define pPin09      GpioPinPwm( B, 1, 1, A )      // PB1, OC1A, PCINT1
00072 #define pPin10      GpioPinPwm( B, 2, 1, B )      // PB2, SS, OC1B, PCINT2
00073 #define pPin11      GpioPinPwm( B, 3, 2, A )      // PB3, MOSI, OC2A, PCINT3
00074 #define pPin12      GpioPin( B, 4 )      // PB4, MISO, PCINT4
00075 #define pPin13      GpioPin( B, 5 )      // PB5, SCK, PCINT5
00076
00077 #define pSDA      pPinA04      // I2C SDA
00078 #define pSCL      pPinA05      // I2C SCL
00079
00080 #define pSS      pPin10      // SPI SS
00081 #define pMOSI      pPin11      // SPI MOSI
00082 #define pMISO      pPin12      // SPI MISO
00083 #define pSCK      pPin13      // SPI SCK
00084
00085
00086 #endif

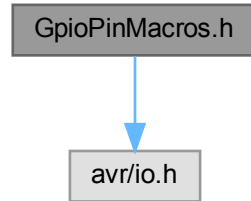
```

10.11 GpioPinMacros.h File Reference

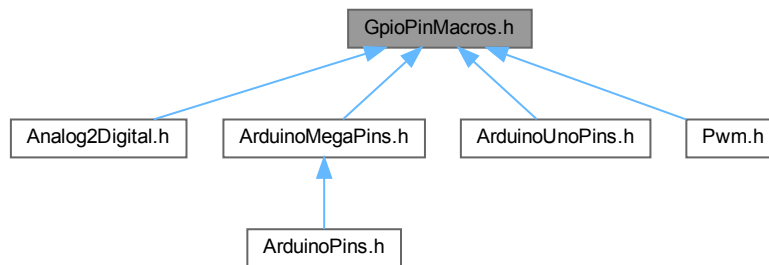
This file contains the primary macros for naming and manipulating GPIO pin names.

```
#include <avr/io.h>
```

Include dependency graph for GpioPinMacros.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [GpioPinVariable](#)

This class defines a type that can encode a GPIO pin as a variable. Read the section on [GPIO Pin Variables] (GPIO pin variables) to understand how to use this class.

Macros

- #define [GpioPin](#)(portLtr, pinNbr)
Primary macro-function for defining a GPIO pin name.
- #define [GpioPinAnalog](#)(portLtr, pinNbr, adcNbr)
Secondary macro-function for defining a GPIO pin name for GPIO pins that support analog conversion.
- #define [GpioPinPwm](#)(portLtr, pinNbr, timer, chan)
Secondary macro-function for defining a GPIO pin name for GPIO pins that support PWM output.
- #define [isGpioPinModeOutput](#)(pinName)

- *Test if the mode of the GPIO pin is output (i.e., the corresponding DDRn bit is set).*
- #define `isGpioPinModeInput`(pinName)
 - *Test if the mode of the GPIO pin is input (i.e., the corresponding DDRn is clear).*
- #define `setGpioPinModeOutput`(pinName)
 - *Set the mode of the GPIO pin to output (i.e., set the corresponding DDRn bit).*
- #define `setGpioPinModeInput`(pinName)
 - *Set the mode of the GPIO pin to input (i.e., clear the corresponding DDRn and PORTn bits).*
- #define `setGpioPinModeInputPullup`(pinName)
 - *Set the mode of the GPIO pin to input with pullup (i.e., clear the corresponding DDRn bit and set the PORTn bit).*
- #define `readGpioPinDigital`(pinName)
 - *Read the value of the GPIO pin (i.e., return the value of corresponding the PINn bit).*
- #define `writeGpioPinDigital`(pinName, val)
 - *Write a value the GPIO pin (i.e., set or clear the corresponding the PORTn bit).*
- #define `setGpioPinHigh`(pinName)
 - *Write a 1 to the GPIO pin (i.e., set the corresponding the PORTn bit).*
- #define `setGpioPinLow`(pinName)
 - *Write a 0 the GPIO pin (i.e., clear the corresponding the PORTn bit).*
- #define `getGpioDDR`(pinName)
 - *Get the DDRn corresponding to this GPIO pin.*
- #define `getGpioPORT`(pinName)
 - *Get the PORTn corresponding to this GPIO pin.*
- #define `getGpioPIN`(pinName)
 - *Get the bit number corresponding to this GPIO pin.*
- #define `getGpioMASK`(pinName)
 - *Get the bit mask corresponding to this GPIO pin.*
- #define `getGpioADC`(pinName)
 - *Get the ADC channel corresponding to this GPIO pin, assuming it is an ADC capable GPIO pin.*
- #define `getGpioOCR`(pinName)
 - *Get the OCR register corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.*
- #define `getGpioCOM`(pinName)
 - *Get the COM bit name corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.*
- #define `getGpioTCCR`(pinName)
 - *Get the TCCR register corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.*
- #define `makeGpioVarFromGpioPin`(pinName)
 - *Create a GPIO pin variable of type `GpioPinVariable` from a GPIO pin macro.*
- #define `makeGpioVarFromGpioPinAnalog`(pinName)
 - *Create a GPIO pin variable of type `GpioPinVariable` that can be used for analog-to-digital reading from a GPIO pin macro.*
- #define `makeGpioVarFromGpioPinPwm`(pinName)
 - *Create a GPIO pin variable of type `GpioPinVariable` that can be used for PWM from a GPIO pin macro.*

Enumerations

- enum { `kDigitalLow` = 0 , `kDigitalHigh` = 1 }
 - *Constants for digital values representing LOW and HIGH.*

Functions

- bool `isGpioPinModeOutputV` (const [GpioPinVariable](#) &pinVar)
Test if the mode of the GPIO pin is output (i.e., the corresponding DDRn bit is set).
- bool `isGpioPinModeInputV` (const [GpioPinVariable](#) &pinVar)
Test if the mode of the GPIO pin is input (i.e., the corresponding DDRn is clear).
- void `setGpioPinModeOutputV` (const [GpioPinVariable](#) &pinVar)
Set the mode of the GPIO pin to output (i.e., set the corresponding DDRn bit).
- void `setGpioPinModeInputV` (const [GpioPinVariable](#) &pinVar)
Set the mode of the GPIO pin to input (i.e., clear the corresponding DDRn and PORTn bits).
- void `setGpioPinModeInputPullupV` (const [GpioPinVariable](#) &pinVar)
Set the mode of the GPIO pin to input with pullup (i.e., clear the corresponding DDRn bit and set the PORTn bit).
- bool `readGpioPinDigitalV` (const [GpioPinVariable](#) &pinVar)
Read the value of the GPIO pin (i.e., return the value of corresponding the PINn bit).
- void `writeGpioPinDigitalV` (const [GpioPinVariable](#) &pinVar, bool value)
Write a value the GPIO pin (i.e., set or clear the corresponding the PORTn bit).
- void `setGpioPinHighV` (const [GpioPinVariable](#) &pinVar)
Write a 1 to the GPIO pin (i.e., set the corresponding the PORTn bit).
- void `setGpioPinLowV` (const [GpioPinVariable](#) &pinVar)
Write a 0 to the GPIO pin (i.e., clear the corresponding the PORTn bit).

10.11.1 Detailed Description

This file contains the primary macros for naming and manipulating GPIO pin names.

Normally you do not include this file directly. Instead include either [ArduinoPins.h](#), which will automatically include this file.

10.11.2 Macro Definition Documentation

10.11.2.1 `getGpioADC`

```
#define getGpioADC(  
    pinName )
```

Get the ADC channel corresponding to this GPIO pin, assuming it is an ADC capable GPIO pin.

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a number between 0-7 (for ATmega328) or between 0-15 (for ATmega2560).

10.11.2.2 getGpioCOM

```
#define getGpioCOM(  
    pinName )
```

Get the COM bit name corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.

- *pinName* a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a COMn[A/B]1 bit name (e.g., COM2B1)

10.11.2.3 getGpioDDR

```
#define getGpioDDR(  
    pinName )
```

Get the DDRn corresponding to this GPIO pin.

- *pinName* a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a DDRn register name (e.g., DDRB)

10.11.2.4 getGpioMASK

```
#define getGpioMASK(  
    pinName )
```

Get the bit mask corresponding to this GPIO pin.

- *pinName* a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a byte-sized bitmask

10.11.2.5 getGpioOCR

```
#define getGpioOCR(  
    pinName )
```

Get the OCR register corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a OCRn[A/B] register name (e.g., OCR2B)

10.11.2.6 getGpioPIN

```
#define getGpioPIN(  
    pinName )
```

Get the bit number corresponding to this GPIO pin.

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a number between 0 and 7

10.11.2.7 getGpioPORT

```
#define getGpioPORT(  
    pinName )
```

Get the PORTn corresponding to this GPIO pin.

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a PORTn register name (e.g., PORTB)

10.11.2.8 getGpioTCCR

```
#define getGpioTCCR(  
    pinName )
```

Get the TCCR register corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.

- *pinName* a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a TCCTn[A/B] register name (e.g., TCCR2B)

10.11.2.9 GpioPin

```
#define GpioPin(  
    portLtr,  
    pinNbr )
```

Primary macro-function for defining a GPIO pin name.

- *portLtr* an uppercase letter identifying the port (e.g., A, B, C, ...) the GPIO pin belongs to.
- *pinNbr* a number between 0 and 7 identifying the bit on that port that corresponds to the GPIO pin.

10.11.2.10 GpioPinAnalog

```
#define GpioPinAnalog(  
    portLtr,  
    pinNbr,  
    adcNbr )
```

Secondary macro-function for defining a GPIO pin name for GPIO pins that support analog conversion.

- *portLtr* an uppercase letter identifying the port (e.g., A, B, C, ...) the GPIO pin belongs to.
- *pinNbr* a number between 0 and 7 identifying the bit on that port that corresponds to the GPIO pin.
- *adcNbr* a number representing the ADC converter channel corresponding to this GPIO pin (0-7 for `ArduinoUno`; 0-15 for `ArduinoMega`)

10.11.2.11 GpioPinPwm

```
#define GpioPinPwm(  
    portLtr,  
    pinNbr,  
    timer,  
    chan )
```

Secondary macro-function for defining a GPIO pin name for GPIO pins that support PWM output.

- `portLtr` an uppercase letter identifying the port (e.g., A, B, C, ...) the GPIO pin belongs to.
- `pinNbr` a number between 0 and 7 identifying the bit on that port that corresponds to the GPIO pin.
- `timer` a number representing the timer number associated with the PWM function on this GPIO pin.
- `chan` a letter (A, B, or C) representing the channel on the timer associated with the PWM function on this GPIO pin.

10.11.2.12 isGpioPinModeInput

```
#define isGpioPinModeInput(  
    pinName )
```

Test if the mode of the GPIO pin is input (i.e., the corresponding DDRn is clear).

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.11.2.13 isGpioPinModeOutput

```
#define isGpioPinModeOutput(  
    pinName )
```

Test if the mode of the GPIO pin is output (i.e., the corresponding DDRn bit is set).

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.11.2.14 makeGpioVarFromGpioPin

```
#define makeGpioVarFromGpioPin(  
    pinName )
```

Create a GPIO pin variable of type [GpioPinVariable](#) from a GPIO pin macro.

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

a [GpioPinVariable](#).

10.11.2.15 makeGpioVarFromGpioPinAnalog

```
#define makeGpioVarFromGpioPinAnalog(  
    pinName )
```

Create a GPIO pin variable of type [GpioPinVariable](#) that can be used for analog-to-digital reading from a GPIO pin macro.

- `pinName` a GPIO pin name macro generated by [GpioPinAnalog\(\)](#).

Returns

a [GpioPinVariable](#) that can be used for analog-to-digital reading.

10.11.2.16 makeGpioVarFromGpioPinPwm

```
#define makeGpioVarFromGpioPinPwm(  
    pinName )
```

Create a GPIO pin variable of type [GpioPinVariable](#) that can be used for PWM from a GPIO pin macro.

- `pinName` a GPIO pin name macro generated by [GpioPinPwm\(\)](#).

Returns

a [GpioPinVariable](#) that can be used for PWM.

10.11.2.17 readGpioPinDigital

```
#define readGpioPinDigital(  
    pinName )
```

Read the value of the GPIO pin (i.e., return the value of corresponding the PINn bit).

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

0 (false) or a non-zero (true) value

10.11.2.18 setGpioPinHigh

```
#define setGpioPinHigh(  
    pinName )
```

Write a 1 to the GPIO pin (i.e., set the corresponding the PORTn bit).

- *pinName* a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.11.2.19 setGpioPinLow

```
#define setGpioPinLow(  
    pinName )
```

Write a 0 the GPIO pin (i.e., clear the corresponding the PORTn bit).

- *pinName* a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.11.2.20 setGpioPinModeInput

```
#define setGpioPinModeInput(  
    pinName )
```

Set the mode of the GPIO pin to input (i.e., clear the corresponding DDRn and PORTn bits).

- *pinName* a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.11.2.21 setGpioPinModeInputPullup

```
#define setGpioPinModeInputPullup(  
    pinName )
```

Set the mode of the GPIO pin to input with pullup (i.e., clear the corresponding DDRn bit and set the PORTn bit).

- *pinName* a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.11.2.22 setGpioPinModeOutput

```
#define setGpioPinModeOutput (
    pinName )
```

Set the mode of the GPIO pin to output (i.e., set the corresponding DDRn bit).

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.11.2.23 writeGpioPinDigital

```
#define writeGpioPinDigital(
    pinName,
    val )
```

Write a value the GPIO pin (i.e., set or clear the corresponding the PORTn bit).

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).
- `val` the value to be written: 0 means to clear the GPIO pin; any other value means to set it.

10.11.3 Enumeration Type Documentation

10.11.3.1 anonymous enum

```
anonymous enum
```

Constants for digital values representing LOW and HIGH.

Enumerator

<code>kDigitalLow</code>	Value representing digital LOW.
<code>kDigitalHigh</code>	Value representing digital HIGH.

10.11.4 Function Documentation

10.11.4.1 isGpioPinModeInputV()

```
bool isGpioPinModeInputV (
    const GpioPinVariable & pinVar ) [inline]
```

Test if the mode of the GPIO pin is input (i.e., the corresponding DDRn is clear).

- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).

10.11.4.2 isGpioPinModeOutputV()

```
bool isGpioPinModeOutputV (  
    const GpioPinVariable & pinVar ) [inline]
```

Test if the mode of the GPIO pin is output (i.e., the corresponding DDRn bit is set).

- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).

10.11.4.3 readGpioPinDigitalV()

```
bool readGpioPinDigitalV (  
    const GpioPinVariable & pinVar ) [inline]
```

Read the value of the GPIO pin (i.e., return the value of corresponding the PINn bit).

- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).

Returns

0 (false) or 1 (true)

10.11.4.4 setGpioPinHighV()

```
void setGpioPinHighV (  
    const GpioPinVariable & pinVar ) [inline]
```

Write a 1 to the GPIO pin (i.e., set the corresponding the PORTn bit).

- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).

10.11.4.5 setGpioPinLowV()

```
void setGpioPinLowV (  
    const GpioPinVariable & pinVar ) [inline]
```

Write a 0 to the GPIO pin (i.e., clear the corresponding the PORTn bit).

- `pinVar` aa GPIO pin variable of type [GpioPinVariable](#).

10.11.4.6 setGpioPinModeInputPullupV()

```
void setGpioPinModeInputPullupV (
    const GpioPinVariable & pinVar ) [inline]
```

Set the mode of the GPIO pin to input with pullup (i.e., clear the corresponding DDRn bit and set the PORTn bit).

- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).

10.11.4.7 setGpioPinModeInputV()

```
void setGpioPinModeInputV (
    const GpioPinVariable & pinVar ) [inline]
```

Set the mode of the GPIO pin to input (i.e., clear the corresponding DDRn and PORTn bits).

- `pinVar` a GPIO pin name variable of type [GpioPinVariable](#).

10.11.4.8 setGpioPinModeOutputV()

```
void setGpioPinModeOutputV (
    const GpioPinVariable & pinVar ) [inline]
```

Set the mode of the GPIO pin to output (i.e., set the corresponding DDRn bit).

- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).

10.11.4.9 writeGpioPinDigitalV()

```
void writeGpioPinDigitalV (
    const GpioPinVariable & pinVar,
    bool value ) [inline]
```

Write a value the GPIO pin (i.e., set or clear the corresponding the PORTn bit).

- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).
- `val` the value to be written: 0 means to clear the GPIO pin; any other value means to set it.

10.12 GpioPinMacros.h

[Go to the documentation of this file.](#)

```

00001 /*
00002     GpioPinMacros.h - Macros for naming and manipulating Arduino pins.
00003     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007     This program is free software: you can redistribute it and/or modify
00008     it under the terms of the GNU General Public License as published by
00009     the Free Software Foundation, either version 3 of the License, or
00010     (at your option) any later version.
00011
00012     This program is distributed in the hope that it will be useful,
00013     but WITHOUT ANY WARRANTY; without even the implied warranty of
00014     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015     GNU General Public License for more details.
00016
00017     You should have received a copy of the GNU General Public License
00018     along with this program. If not, see <http://www.gnu.org/licenses/>.
00019 */
00020
00021 #ifndef GpioPinMacros_h
00022 #define GpioPinMacros_h
00023
00024 #include <avr/io.h>
00025
00026 enum
00027 {
00028     kDigitalLow = 0,
00029     kDigitalHigh = 1
00030 };
00031
00032 /*
00033     These macros are implementation details for the port naming macros and are not intended
00034     for end-users. These are required to make the macros work due to the
00035     reparsing and resubstitution rules of the C/C++ preprocessor.
00036 */
00037
00038 #define _GpioPin( ddr, port, pin, nbr ) \
00039     ddr, port, pin, nbr, -1, 0, -1, 0
00040
00041 #define _GpioPinAnalog( ddr, port, pin, nbr, adc ) \
00042     ddr, port, pin, nbr, adc, 0, -1, 0
00043
00044 #define _GpioPinPwm( ddr, port, pin, nbr, ocr, com, tccr ) \
00045     ddr, port, pin, nbr, -1, ocr, com, tccr
00046
00047 #define _isGpioPinModeOutput( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00048     ( ddr & (1<nbr) )
00049
00050 #define _isGpioPinModeInput( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00051     (!( ddr & (1<nbr) ))
00052
00053 #define _setGpioPinModeOutput( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00054     ddr |= (1<nbr)
00055
00056 #define _setGpioPinModeInput( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00057     ddr &= ~(1<nbr), port &= \
00058     ~(1<nbr)
00059
00060 #define _setGpioPinModeInputPullup( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00061     ddr &= ~(1<nbr), port |= \
00062     (1<nbr)
00063
00064 #define _readGpioPinDigital( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00065     ( pin & (1<nbr) )
00066
00067 #define _writeGpioPinDigital( ddr, port, pin, nbr, adc, ocr, com, tccr, value ) \
00068     do { if (value) port |= (1<nbr); else port &= ~(1<nbr); } while ( \
00069     0 )
00070
00071 #define _setGpioPinHigh( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00072     port |= (1<nbr)

```

```

00091
00092 #define _setGpioPinLow( ddr, port, pin, nbr, adc, ocr, com, tccr )      port &= ~(1<nbr)
00093
00094 #define _getGpioDDR( ddr, port, pin, nbr, adc, ocr, com, tccr )      ddr
00095
00096 #define _getGpioPORT( ddr, port, pin, nbr, adc, ocr, com, tccr )      port
00097
00098 #define _getGpioPIN( ddr, port, pin, nbr, adc, ocr, com, tccr )      pin
00099
00100 #define _getGpioMASK( ddr, port, pin, nbr, adc, ocr, com, tccr )      (1<nbr)
00101
00102 #define _getGpioADC( ddr, port, pin, nbr, adc, ocr, com, tccr )      adc
00103
00104 #define _getGpioOCR( ddr, port, pin, nbr, adc, ocr, com, tccr )      ocr
00105
00106 #define _getGpioCOM( ddr, port, pin, nbr, adc, ocr, com, tccr )      com
00107
00108 #define _getGpioTCCR( ddr, port, pin, nbr, adc, ocr, com, tccr )      tccr
00109
00110
00111
00112
00113
00114 /*
00115     These macros are for end-users to name GPIO pins and manipulate GPIO pin name macros.
00116 */
00117
00118 */
00119
00120
00121
00122 #define GpioPin( portLtr, pinNbr )      \
00123     _GpioPin( DDR##portLtr, PORT##portLtr, PIN##portLtr, pinNbr )
00124
00125
00126 #define GpioPinAnalog( portLtr, pinNbr, adcNbr )      \
00127     _GpioPinAnalog( DDR##portLtr, PORT##portLtr, PIN##portLtr, pinNbr,
00128     adcNbr )
00129
00130
00131 #define GpioPinPwm( portLtr, pinNbr, timer, chan )      \
00132     _GpioPinPwm( DDR##portLtr, PORT##portLtr, PIN##portLtr, pinNbr, OCR##timer##chan, COM##timer##chan##1,
00133     TCCR##timer##A )
00134
00135
00136
00137
00138 #define isGpioPinModeOutput( pinName )      _isGpioPinModeOutput(
00139     pinName )
00140
00141
00142
00143 #define isGpioPinModeInput( pinName )      _isGpioPinModeInput(
00144     pinName )
00145
00146
00147
00148 #define setGpioPinModeOutput( pinName )      _setGpioPinModeOutput(
00149     pinName )
00150
00151
00152
00153 #define setGpioPinModeInput( pinName )      _setGpioPinModeInput(
00154     pinName )
00155
00156
00157
00158 #define setGpioPinModeInputPullup( pinName )      _setGpioPinModeInputPullup( pinName )
00159
00160
00161
00162 #define readGpioPinDigital( pinName )      _readGpioPinDigital(
00163     pinName )
00164
00165
00166
00167 #define writeGpioPinDigital( pinName, val )      _writeGpioPinDigital(
00168     pinName, val )
00169
00170
00171
00172
00173
00174
00175
00176
00177
00178
00179
00180
00181
00182
00183
00184
00185
00186
00187
00188
00189
00190
00191
00192
00193
00194
00195
00196
00197
00198
00199
00200
00201
00202
00203
00204
00205
00206
00207
00208
00209
00210
00211
00212
00213
00214
00215
00216
00217
00218
00219
00220
00221
00222
00223
00224
00225
00226
00227
00228
00229
00230
00231
00232
00233
00234
00235
00236
00237
00238
00239
00240
00241
00242
00243
00244
00245
00246
00247
00248
00249
00250
00251

```

```

00260 #define setGpioPinHigh( pinName )                _setGpioPinHigh( pinName )
00261
00262
00263
00272 #define setGpioPinLow( pinName )                _setGpioPinLow( pinName )
00273
00274
00275
00286 #define getGpioDDR( pinName )                  _getGpioDDR( pinName )
00287
00288
00289
00300 #define getGpioPORT( pinName )                _getGpioPORT( pinName )
00301
00302
00303
00314 #define getGpioPIN( pinName )                _getGpioPIN( pinName )
00315
00316
00317
00328 #define getGpioMASK( pinName )              _getGpioMASK( pinName )
00329
00330
00331
00342 #define getGpioADC( pinName )                _getGpioADC( pinName )
00343
00344
00345
00356 #define getGpioOCR( pinName )                _getGpioOCR( pinName )
00357
00358
00359
00370 #define getGpioCOM( pinName )                _getGpioCOM( pinName )
00371
00372
00373
00384 #define getGpioTCCR( pinName )              _getGpioTCCR( pinName )
00385
00386
00387
00388
00389 /*****
00390
00391 */
00392 * Support for GPIO pin variables
00393 *
00394 */
00395
00396
00397 typedef volatile uint8_t* Gpio8Ptr;
00398 typedef volatile uint16_t* Gpio16Ptr;
00399
00424 class GpioPinVariable
00425 {
00426 public:
00427
00428     GpioPinVariable()
00429     : mDdr( 0 ), mPort( 0 ), mPin( 0 ), mOcr( 0 ), mTccr( 0 ), mCom( 0xFF ),
00430       mNbr( 0xFF ), mAdc( 0xFF )
00431     {}
00432
00433     GpioPinVariable( Gpio8Ptr ddr, Gpio8Ptr port, Gpio8Ptr pin, int8_t nbr )
00434     : mDdr( ddr ), mPort( port ), mPin( pin ), mOcr( 0 ), mTccr( 0 ), mCom( 0xFF ),
00435       mNbr( static_cast<uint8_t>(nbr) ), mAdc( 0xFF )
00436     {}
00437
00438     GpioPinVariable( Gpio8Ptr ddr, Gpio8Ptr port, Gpio8Ptr pin, int8_t nbr, int8_t adc )
00439     : mDdr( ddr ), mPort( port ), mPin( pin ), mOcr( 0 ), mTccr( 0 ), mCom( 0xFF ),
00440       mNbr( static_cast<uint8_t>(nbr) ), mAdc( static_cast<uint8_t>(adc) )
00441     {}
00442
00443     GpioPinVariable( Gpio8Ptr ddr, Gpio8Ptr port, Gpio8Ptr pin, int8_t nbr, Gpio16Ptr ocr, Gpio8Ptr tccr,
00444       int8_t com )
00445     : mDdr( ddr ), mPort( port ), mPin( pin ), mOcr( ocr ), mTccr( tccr ), mCom( com ),
00446       mNbr( static_cast<uint8_t>(nbr) ), mAdc( 0xFF )
00447     {}
00448
00449     Gpio8Ptr ddr() const
00450     { return mDdr; }
00451
00452     Gpio8Ptr port() const

```

```

00454     { return mPort; }
00455
00457     Gpio8Ptr pin() const
00458     { return mPin; }
00459
00461     Gpio16Ptr ocr() const
00462     { return mOcr; }
00463
00465     Gpio8Ptr tccr() const
00466     { return mTccr; }
00467
00469     uint8_t bitNbr() const
00470     { return mNbr; }
00471
00473     uint8_t com() const
00474     { return mCom; }
00475
00477     uint8_t adcNbr() const
00478     { return mAdc; }
00479
00480
00481 private:
00482
00483     Gpio8Ptr      mDdr;
00484     Gpio8Ptr      mPort;
00485     Gpio8Ptr      mPin;
00486     Gpio16Ptr     mOcr;
00487     Gpio8Ptr      mTccr;
00488     uint8_t       mCom;
00489     uint8_t       mNbr;
00490     uint8_t       mAdc;
00491 };
00492
00493
00494
00495
00496
00497 #define _makeGpioVarFromGpioPin( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00498                                     GpioPinVariable( &(ddr), &(port), &(pin), nbr )
00499
00500 #define _makeGpioVarFromGpioPinAnalog( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00501                                     GpioPinVariable( &(ddr), &(port), &(pin), nbr, adc )
00502
00503 #define _makeGpioVarFromGpioPinPwm( ddr, port, pin, nbr, adc, ocr, com, tccr ) \
00504                                     GpioPinVariable( &(ddr), &(port), &(pin), nbr, &(ocr),
&(tccr), com )
00505
00506
00507
00518 #define makeGpioVarFromGpioPin( pinName )           _makeGpioVarFromGpioPin( pinName )
00519
00520
00531 #define makeGpioVarFromGpioPinAnalog( pinName )     _makeGpioVarFromGpioPinAnalog( pinName )
00532
00533
00544 #define makeGpioVarFromGpioPinPwm( pinName )        _makeGpioVarFromGpioPinPwm( pinName )
00545
00546
00547
00548
00549
00556 inline bool isGpioPinModeOutputV( const GpioPinVariable& pinVar )
00557 {
00558     return *(pinVar.ddr()) & ( 1 « pinVar.bitNbr() );
00559 }
00560
00561
00568 inline bool isGpioPinModeInputV( const GpioPinVariable& pinVar )
00569 {
00570     return !( *(pinVar.ddr()) & ( 1 « pinVar.bitNbr() ) );
00571 }
00572
00573
00574
00575
00576
00585 inline void setGpioPinModeOutputV( const GpioPinVariable& pinVar )
00586 {
00587     *(pinVar.ddr()) |= ( 1 « pinVar.bitNbr() );
00588 }
00589

```

```

00590
00591
00592
00601 inline void setGpioPinModeInputV( const GpioPinVariable& pinVar )
00602 {
00603     *(pinVar.ddd()) &= ~( 1 « pinVar.bitNbr() );
00604     *(pinVar.port()) &= ~( 1 « pinVar.bitNbr() );
00605 }
00606
00607
00608
00617 inline void setGpioPinModeInputPullupV( const GpioPinVariable& pinVar )
00618 {
00619     *(pinVar.ddd()) &= ~( 1 « pinVar.bitNbr() );
00620     *(pinVar.port()) |= ( 1 « pinVar.bitNbr() );
00621 }
00622
00623
00624
00635 inline bool readGpioPinDigitalV( const GpioPinVariable& pinVar )
00636 {
00637     return *(pinVar.pin()) & ( 1 « pinVar.bitNbr() );
00638 }
00639
00640
00641
00642
00652 inline void writeGpioPinDigitalV( const GpioPinVariable& pinVar, bool value )
00653 {
00654     if ( value )
00655     {
00656         *(pinVar.port()) |= ( 1 « pinVar.bitNbr() );
00657     }
00658     else
00659     {
00660         *(pinVar.port()) &= ~( 1 « pinVar.bitNbr() );
00661     }
00662 }
00663
00664
00665
00674 inline void setGpioPinHighV( const GpioPinVariable& pinVar )
00675 {
00676     *(pinVar.port()) |= ( 1 « pinVar.bitNbr() );
00677 }
00678
00679
00680
00681
00690 inline void setGpioPinLowV( const GpioPinVariable& pinVar )
00691 {
00692     *(pinVar.port()) &= ~( 1 « pinVar.bitNbr() );
00693 }
00694
00695
00696
00697 #endif

```

10.13 I2cLcd.h File Reference

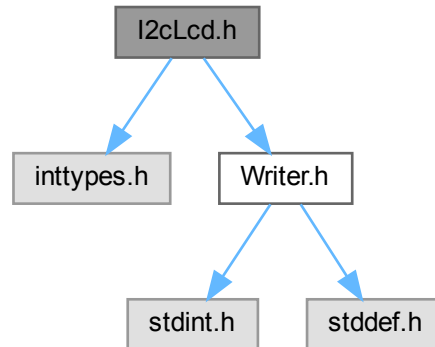
This file defines a class that provides a high-level interface to an LCD offering an I2C interface. The most common variant of this is HD44780U controlled LCD driven via an MCP23017 that offers an I2C interface (such LCDs are available from Adafruit and SparkFun). To use this class you must also use and properly initialize the I2C Master package from [I2cMaster.h](#).

```

#include <inttypes.h>
#include "Writer.h"

```

Include dependency graph for I2cLcd.h:



Classes

- class [I2cLcd](#)

This class provides a high-level interface via I2C to an LCD such as those offered by AdaFruit and SparkFun. Specifically, it communicates via I2C with an MCP23017 that drives an HD44780U controlling an LCD. It also lets you detect button presses on the 5-button keypad generally associated with such devices.

10.13.1 Detailed Description

This file defines a class that provides a high-level interface to an LCD offering an I2C interface. The most common variant of this is HD44780U controlled LCD driven via an MCP23017 that offers an I2C interface (such LCDs are available from Adafruit and SparkFun). To use this class you must also use and properly initialize the I2C Master package from [I2cMaster.h](#).

To use these features, include [I2cLcd.h](#) in your source code and link against I2cLcd.cpp and I2cMaster.cpp.

10.14 I2cLcd.h

[Go to the documentation of this file.](#)

```

00001 /*
00002     I2cLcd.h - Tools for using an I2C-based LCD such as the
00003     Adafruit RGB 16x2 LCD Shield .
00004     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00005     This is part of the AVRTools library.
00006     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00007
00008     This program is free software: you can redistribute it and/or modify
00009     it under the terms of the GNU General Public License as published by
00010     the Free Software Foundation, either version 3 of the License, or
00011     (at your option) any later version.
00012

```

```
00013      This program is distributed in the hope that it will be useful,
00014      but WITHOUT ANY WARRANTY; without even the implied warranty of
00015      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00016      GNU General Public License for more details.
00017
00018      You should have received a copy of the GNU General Public License
00019      along with this program. If not, see <http://www.gnu.org/licenses/>.
00020  */
00021
00022
00037 #ifndef I2cLcd_h
00038 #define I2cLcd_h
00039
00040 #include <inttypes.h>
00041 #include "Writer.h"
00042
00043
00044
00045
00057 class I2cLcd : public Writer
00058 {
00059 public:
00060
00064     enum
00065     {
00066         kButton_Select    = 0x01,
00067         kButton_Right     = 0x02,
00068         kButton_Down      = 0x04,
00069         kButton_Up        = 0x08,
00070         kButton_Left      = 0x10
00071     };
00072
00073
00077     enum
00078     {
00079         kBacklight_Red    = 0x1,
00080         kBacklight_Yellow = 0x3,
00081         kBacklight_Green  = 0x2,
00082         kBacklight_Teal   = 0x6,
00083         kBacklight_Blue   = 0x4,
00084         kBacklight_Violet = 0x5,
00085         kBacklight_White  = 0x7
00086     };
00087
00088
00092     I2cLcd();
00093
00094
00102     int  init();
00103
00104
00108     void clear();
00109
00110
00114     void home();
00115
00116
00122     void displayTopRow( const char* str );
00123
00124
00130     void displayBottomRow( const char* str );
00131
00132
00136     void clearTopRow();
00137
00138
00142     void clearBottomRow();
00143
00144
00145
00149     void displayOff();
00150
00151
00155     void displayOn();
00156
00157
00161     void blinkOff();
00162
00163
00167     void blinkOn();
00168
```



```

00169
00173     void cursorOff();
00174
00175
00179     void cursorOn();
00180
00181
00185     void scrollDisplayLeft();
00186
00187
00191     void scrollDisplayRight();
00192
00193
00197     void autoscrollOn();
00198
00199
00203     void autoscrollOff();
00204
00205
00212     void setCursor( uint8_t row, uint8_t col );
00213
00214
00222     int setBacklight( uint8_t color );
00223
00224
00230     void command( uint8_t cmd );
00231
00232
00240     uint8_t readButtons();
00241
00242
00243
00252     virtual size_t write( char c );
00253
00254
00263     virtual size_t write( const char* str );
00264
00265
00275     virtual size_t write( const char* buffer, size_t size );
00276
00277
00287     virtual size_t write( const uint8_t* buffer, size_t size );
00288
00289
00294     virtual void flush();
00295
00296
00297 private:
00298
00299     enum
00300     {
00301         kWriteFourBitsSendChar = 0,
00302         kWriteFourBitsSendCommand = 1
00303     };
00304
00305     int initMCP23017();
00306     int initHD44780U();
00307     size_t write( uint8_t value );
00308     int writeFourBitsToLcd( uint8_t value, uint8_t gpioB );
00309     int sendCharOrCmdToLcd( uint8_t value, bool isCommand );
00310
00311     int sendCommand( uint8_t cmd )
00312     {
00313         return sendCharOrCmdToLcd( cmd, kWriteFourBitsSendCommand );
00314     }
00315
00316     int sendCharToDisplay( uint8_t value )
00317     {
00318         return sendCharOrCmdToLcd( value, kWriteFourBitsSendChar );
00319     }
00320
00321     uint8_t          mDisplayControl;
00322     uint8_t          mDisplayMode;
00323     uint8_t          mCurrLine;
00324     volatile uint8_t mI2cStatus;
00325 };
00326
00327 #endif

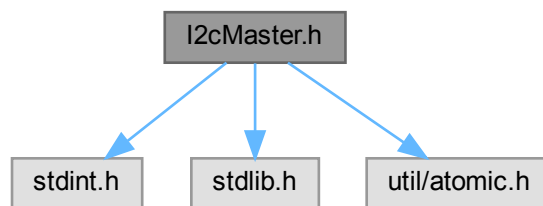
```

10.15 I2cMaster.h File Reference

This file provides functions that interface to the TWI (two-wire serial interface) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. Include this file if you want your application will operate in Master mode as defined in the I2C protocol.

```
#include <stdint.h>
#include <stdlib.h>
#include <util/atomic.h>
```

Include dependency graph for I2cMaster.h:



Namespaces

- namespace [I2cMaster](#)

This namespace bundles the I2C-protocol-based interface to the TWI hardware. It provides logical cohesion for functions implement the Master portions of the I2C protocol and prevents namespace collisions.

Enumerations

- enum [I2cMaster::I2cBusSpeed](#) { [I2cMaster::kI2cBusSlow](#) , [I2cMaster::kI2cBusFast](#) }
This enum lists I2C bus speed configurations.
- enum [I2cMaster::I2cStatusCodes](#) { [I2cMaster::kI2cCompletedOk](#) = 0x00 , [I2cMaster::kI2cError](#) = 0x01 , [I2cMaster::kI2cNotStarted](#) = 0x02 , [I2cMaster::kI2cInProgress](#) = 0x04 }
This enum lists I2C status codes reported by the various transmit functions.
- enum [I2cMaster::I2cSendErrorCodes](#) { [I2cMaster::kI2cNoError](#) = 0 , [I2cMaster::kI2cErrTxBufferFull](#) = 1 , [I2cMaster::kI2cErrMsgTooLong](#) = 2 , [I2cMaster::kI2cErrNullStatusPtr](#) = 3 , [I2cMaster::kI2cErrWriteWithoutData](#) = 4 , [I2cMaster::kI2cErrReadWithoutStorage](#) = 5 }
This enum lists I2C errors codes that may occur when you try to write a message.
- enum [I2cMaster::I2cPullups](#) { [I2cMaster::kPullupsOff](#) , [I2cMaster::kPullupsOn](#) }
This enum lists the options for controlling the built-in pullups in the TWI hardware.

Functions

- void **I2cMaster::start** (uint8_t speed=**kI2cBusFast**)
Configures the TWI hardware for I2C communications in Master mode. You must call this function before conducting any I2C communications using the functions in this module.
- void **I2cMaster::stop** ()
Terminates the I2C communications using the TWI hardware, and disables the TWI interrupts.
- void **I2cMaster::pullups** (uint8_t set=**kPullupsOn**)
Sets the state of the internal pullups that are part of the TWI hardware.
- bool **I2cMaster::busy** ()
Reports whether the TWI hardware is busy communicating (either transmitting or receiving).
- uint8_t **I2cMaster::writeAsync** (uint8_t address, uint8_t registerAddress, volatile uint8_t *status)
Transmit a single register address (a one-byte message) asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- uint8_t **I2cMaster::writeAsync** (uint8_t address, uint8_t registerAddress, uint8_t data, volatile uint8_t *status)
Transmit a single register address and corresponding single byte of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- uint8_t **I2cMaster::writeAsync** (uint8_t address, uint8_t registerAddress, const char *data, volatile uint8_t *status)
Transmit a single register address and corresponding null-terminated string of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- uint8_t **I2cMaster::writeAsync** (uint8_t address, uint8_t registerAddress, uint8_t *data, uint8_t numberBytes, volatile uint8_t *status)
Transmit a single register address and corresponding buffer of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- uint8_t **I2cMaster::readAsync** (uint8_t address, uint8_t numberBytes, volatile uint8_t *destination, volatile uint8_t *bytesRead, volatile uint8_t *status)
Request to read data from a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports kI2cCompletedOk, the requested data can be read from the receive buffer.
- uint8_t **I2cMaster::readAsync** (uint8_t address, uint8_t registerAddress, uint8_t numberBytes, volatile uint8_t *destination, volatile uint8_t *bytesRead, volatile uint8_t *status)
Request to read data from a specific register on a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports kI2cCompletedOk, the requested data can be read from the receive buffer.
- int **I2cMaster::writeSync** (uint8_t address, uint8_t registerAddress)
Transmit a single register address (a one-byte message) synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int **I2cMaster::writeSync** (uint8_t address, uint8_t registerAddress, uint8_t data)
Transmit a single register address and corresponding single byte of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int **I2cMaster::writeSync** (uint8_t address, uint8_t registerAddress, const char *data)
Transmit a single register address and corresponding null-terminated string of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int **I2cMaster::writeSync** (uint8_t address, uint8_t registerAddress, uint8_t *data, uint8_t numberBytes)
Transmit a single register address and corresponding buffer of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

- int [I2cMaster::readSync](#) (uint8_t address, uint8_t numberBytes, uint8_t *destination)
Request to read data from a device and receive that data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int [I2cMaster::readSync](#) (uint8_t address, uint8_t registerAddress, uint8_t numberBytes, uint8_t *destination)
Request to read data from a specific register on a device and receive that data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

10.15.1 Detailed Description

This file provides functions that interface to the TWI (two-wire serial interface) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. Include this file if you want your application will operate in Master mode as defined in the I2C protocol.

To use these functions, include [I2cMaster.h](#) and link against I2cMaster.cpp.

These interfaces are buffered for both input and output and operate using interrupts associated with the TWI hardware. This means the asynchronous transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated TWI hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit buffer is a ring buffer. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). Receive buffers are provided by the callers of these functions. Note that due to the nature of the I2C protocol, Master I2C "read" operations must still write a command instructing the destination device to send data for the Master to read, and thus "read" operations still utilize the transmit buffer.

The size of the transmit buffer can be set at compile time via macro constants (the receive buffers are provided the corresponding functions are called). The default size of the transmit buffer assumes the maximum transmit message length is 24 bytes and allows 3 out-going messages to be queued. You can change these defaults by defining the macros `I2C_MASTER_MAX_TX_MSG_LEN` to specify the maximum transmit message length and `I2C_MASTER_MAX_TX_MSG_NBR` to specify the maximum number of transmit messages to hold in the buffer. You need to make these define these macros prior to including the file [I2cMaster.h](#), each time it is included. So you should define these using a compiler option (e.g., `-DI2C_MASTER_MAX_TX_MSG_LEN=32 -DI2C_MASTER_MAX_TX_MSG_NBR=5`) to ensure they are consistently defined throughout your project.

This interface assumes your application will operator in I2C Master mode as defined in the I2C protocol. If you wish your application to operate in I2C Slave mode, then instead include [I2cSlave.h](#) and link against I2cSlave.cpp.

Note

Only one of I2cMaster.cpp and I2cSlave.cpp can be linking into your application. These two files install different, incompatible versions of the TWI interrupt function. AVRTools does not support building an application that functions both as a Master and as a Slave under the I2C protocol. This limitation allows the corresponding TWI interrupt functions to be significantly leaner and faster.

10.16 I2cMaster.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      I2cMaster.h - An I2C master library
00003      For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004      This is part of the AVRTools library.
00005      Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007      This program is free software: you can redistribute it and/or modify
00008      it under the terms of the GNU General Public License as published by
00009      the Free Software Foundation, either version 3 of the License, or
00010      (at your option) any later version.
00011
00012      This program is distributed in the hope that it will be useful,
00013      but WITHOUT ANY WARRANTY; without even the implied warranty of
00014      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015      GNU General Public License for more details.
00016
00017      You should have received a copy of the GNU General Public License
00018      along with this program. If not, see <http://www.gnu.org/licenses/>.
00019  */
00020
00021
00068 #ifndef I2cMaster_h
00069 #define I2cMaster_h
00070
00071
00072 #ifdef I2cSlave_h
00073 #error "You cannot use both I2cMaster and I2cSlave in the same application"
00074 #endif
00075
00076
00077 #include <stdint.h>
00078 #include <stdlib.h>
00079
00080 #include <util/atomic.h>
00081
00082
00083 #if defined( DEBUG_I2cMasterBuffer ) || defined( DEBUG_I2cMasterDiary )
00084 #include "USART0.h"
00085 #endif
00086
00087
00088
00089
00090
00091 #ifndef I2C_MASTER_MAX_TX_MSG_LEN
00092 #define I2C_MASTER_MAX_TX_MSG_LEN      24
00093 #endif
00094
00095 #ifndef I2C_MASTER_MAX_TX_MSG_NBR
00096 #define I2C_MASTER_MAX_TX_MSG_NBR      3
00097 #endif
00098
00099 #if I2C_MASTER_MAX_TX_MSG_LEN > 255
00100 #error "I2C_MASTER_MAX_TX_MSG_LEN exceeds size of a uint8_t"
00101 #endif
00102
00103 #if I2C_MASTER_MAX_TX_MSG_NBR > 255
00104 #error "I2C_MASTER_MAX_TX_MSG_NBR exceeds size of a uint8_t"
00105 #endif
00106
00107
00108
00109
00110
00142 namespace I2cMaster
00143 {
00144
00150     enum I2cBusSpeed
00151     {
00152         kI2cBusSlow          = 0,
00153         kI2cBusFast          = 1
00154     };
00155
00160     enum I2cStatusCodes
00161     {

```

```

00162         kI2cCompletedOk                = 0x00,
00163         kI2cError                      = 0x01,
00164         kI2cNotStarted                = 0x02,
00165         kI2cInProgress                = 0x04
00166     };
00167
00168
00172     enum I2cSendErrorCodes
00173     {
00174         kI2cNoError                    = 0,
00175         kI2cErrTxBufferFull            = 1,
00176         kI2cErrMsgTooLong              = 2,
00177         kI2cErrNullStatusPtr           = 3,
00178         kI2cErrWriteWithoutData        = 4,
00179         kI2cErrReadWithoutStorage      = 5
00180     };
00181
00182
00188     enum I2cPullups
00189     {
00190         kPullupsOff                    = 0,
00191         kPullupsOn                     = 1
00192     };
00193
00194
00195
00196
00197
00207     void start( uint8_t speed = kI2cBusFast );
00208
00209
00215     void stop();
00216
00217
00226     void pullups( uint8_t set = kPullupsOn );
00227
00228
00235     bool busy();
00236
00237
00238
00239
00240     // Asynchronous functions
00241
00242
00259     uint8_t writeAsync( uint8_t address, uint8_t registerAddress, volatile uint8_t* status );
00260
00261
00279     uint8_t writeAsync( uint8_t address, uint8_t registerAddress, uint8_t data, volatile uint8_t* status
);
00280
00281
00300     uint8_t writeAsync( uint8_t address, uint8_t registerAddress, const char* data, volatile uint8_t*
status );
00301
00302
00322     uint8_t writeAsync( uint8_t address, uint8_t registerAddress, uint8_t* data, uint8_t numberBytes,
00323                       volatile uint8_t* status );
00324
00325
00326
00347     uint8_t readAsync( uint8_t address, uint8_t numberBytes, volatile uint8_t* destination,
00348                      volatile uint8_t* bytesRead, volatile uint8_t* status );
00349
00350
00374     uint8_t readAsync( uint8_t address, uint8_t registerAddress, uint8_t numberBytes,
00375                      volatile uint8_t* destination, volatile uint8_t* bytesRead,
00376                      volatile uint8_t* status );
00377
00378
00379
00380     // Synchronous
00381
00382
00394     int writeSync( uint8_t address, uint8_t registerAddress );
00395
00396
00409     int writeSync( uint8_t address, uint8_t registerAddress, uint8_t data );
00410
00411
00426     int writeSync( uint8_t address, uint8_t registerAddress, const char* data );

```

```

00427
00428
00445     int writeSync( uint8_t address, uint8_t registerAddress, uint8_t* data, uint8_t numberBytes );
00446
00447
00448
00462     int readSync( uint8_t address, uint8_t numberBytes, uint8_t* destination );
00463
00464
00481     int readSync( uint8_t address, uint8_t registerAddress, uint8_t numberBytes, uint8_t* destination );
00482
00483
00484 #if defined( DEBUG_I2cMasterBuffer ) || defined( DEBUG_I2cMasterDiary )
00485     void setDebugSout( Serial0* s );
00486 #endif
00487
00488 #ifdef DEBUG_I2cMasterBuffer
00489     void dumpBufferContents();
00490 #endif
00491
00492 #ifdef DEBUG_I2cMasterDiary
00493     void clearDebugI2cDiary();
00494     void dumpDebugI2cDiary();
00495 #endif
00496
00497
00498 };
00499
00500
00501 #endif

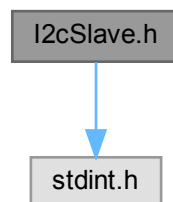
```

10.17 I2cSlave.h File Reference

This file provides functions that interface to the TWI (two-wire serial interface) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. Include this file if you want your application will operate in Slave mode as defined in the I2C protocol.

```
#include <stdint.h>
```

Include dependency graph for I2cSlave.h:



Namespaces

- namespace [I2cSlave](#)

This namespace bundles the I2C-protocol-based interface to the TWI hardware. It provides logical cohesion for functions implement the Slave portions of the I2C protocol and prevents namespace collisions.

Enumerations

- enum [I2cSlave::I2cBusSpeed](#) { [I2cSlave::kI2cBusSlow](#) , [I2cSlave::kI2cBusFast](#) }
This enum lists I2C bus speed configurations.
- enum [I2cSlave::I2cStatusCodes](#) {
 [I2cSlave::kI2cCompletedOk](#) = 0x00 , [I2cSlave::kI2cError](#) = 0x01 , [I2cSlave::kI2cTxPartial](#) = 0x02 ,
 [I2cSlave::kI2cRxOverflow](#) = 0x04 ,
 [I2cSlave::kI2cInProgress](#) = 0x06 }
This enum lists I2C status codes reported by the various transmit functions.
- enum [I2cSlave::I2cPullups](#) { [I2cSlave::kPullupsOff](#) , [I2cSlave::kPullupsOn](#) }
This enum lists the options for controlling the built-in pullups in the TWI hardware.

Functions

- uint8_t [I2cSlave::processI2cMessage](#) (uint8_t *buffer, uint8_t len)
This function must be defined by the user. It is called by the TWI interrupt function installed as part of I2cSlave.cpp whenever it receives a message from the Master. The user should implement this function to respond to the data in the buffer, taking actions and as appropriate returning data to the buffer (for asynchronous transmission to the Master).
- void [I2cSlave::start](#) (uint8_t ownAddress, uint8_t speed=[kI2cBusFast](#), bool answerGeneralCall=false)
Configures the TWI hardware for I2C communications in Slave mode. You must call this function before conducting any I2C communications using the functions in this module.
- void [I2cSlave::stop](#) ()
Terminates the I2C communications using the TWI hardware, and disables the TWI interrupts.
- void [I2cSlave::pullups](#) (uint8_t set=[kPullupsOn](#))
Sets the state of the internal pullups that are part of the TWI hardware.
- bool [I2cSlave::busy](#) ()
Reports whether the TWI hardware is busy communicating (either transmitting or receiving).

10.17.1 Detailed Description

This file provides functions that interface to the TWI (two-wire serial interface) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. Include this file if you want your application will operate in Slave mode as defined in the I2C protocol.

To use these functions, include [I2cSlave.h](#) and link against I2cSlave.cpp.

These interfaces are buffered for receiving and sending data and operate using interrupts associated with the TWI hardware. This means data from the Master is received asynchronously and when reception is complete, a user-supplied function is called. That function has the option of placing data in a buffer to be transmitted asynchronously back to the Master.

The Slave buffer is a simple array. The size of the Slave buffer can be set at compile time via the macro constant `I2C_SLAVE_BUFFER_SIZE`. The default size of the Slave buffer is 32 bytes. You can change the default by defining the macro `I2C_SLAVE_BUFFER_SIZE` prior to including the file [I2cSlave.h](#), each time it is included. So you should define it using a compiler option (e.g., `-DI2C_SLAVE_BUFFER_SIZE=64`) to ensure it is consistently defined throughout your project.

This interface assumes your application will operator in I2C Slave mode as defined in the I2C protocol. If you wish your application to operate in I2C Master mode, then instead include [I2cMaster.h](#) and link against I2cMaster.cpp.

Note

Only one of I2cMaster.cpp and I2cSlave.cpp can be linking into your application. These two files install different, incompatible versions of the TWI interrupt function. AVRTools does not support building an application that functions both as a Master and as a Slave under the I2C protocol. This limitation allows the corresponding TWI interrupt functions to be significantly leaner and faster.

10.18 I2cSlave.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      I2cSlave.h - An I2C slave library
00003      For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004      This is part of the AVRTools library.
00005      Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007      This program is free software: you can redistribute it and/or modify
00008      it under the terms of the GNU General Public License as published by
00009      the Free Software Foundation, either version 3 of the License, or
00010      (at your option) any later version.
00011
00012      This program is distributed in the hope that it will be useful,
00013      but WITHOUT ANY WARRANTY; without even the implied warranty of
00014      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015      GNU General Public License for more details.
00016
00017      You should have received a copy of the GNU General Public License
00018      along with this program. If not, see <http://www.gnu.org/licenses/>.
00019  */
00020
00021
00022
00023
00024
00059  #ifndef I2cSlave_h
00060  #define I2cSlave_h
00061
00062
00063  #ifdef I2cMaster_h
00064  #error "You cannot use both I2cMaster and I2cSlave in the same application"
00065  #endif
00066
00067
00068  #include <stdint.h>
00069
00070  #ifdef DEBUG_I2cSlaveDiary
00071  #include "USART0.h"
00072  #endif
00073
00074
00075
00076  #ifndef I2C_SLAVE_BUFFER_SIZE
00077  #define I2C_SLAVE_BUFFER_SIZE 32
00078  #endif
00079
00080  #if I2C_SLAVE_BUFFER_SIZE > 255
00081  #error "I2C_SLAVE_BUFFER_SIZE exceeds size of a uint8_t"
00082  #endif
00083
00084
00085
00086
00087
00107  namespace I2cSlave
00108  {
00109
00110
00138      uint8_t processI2cMessage( uint8_t* buffer, uint8_t len );
00139
00140
00141
00142
00148      enum I2cBusSpeed
00149      {
00150          kI2cBusSlow = 0,
00151          kI2cBusFast = 1
00152      };
00153
00154
00155
00159      enum I2cStatusCodes
00160      {
00161  /*
00162          kI2cCompletedOk = 0x00,
00163          kI2cNotStarted = 0x01,
00164          kI2cInProgress = 0x02,

```

```

00165         kI2cError                = 0x04,
00166         kI2cBusError              = 0x07
00167     */
00168         kI2cCompletedOk           = 0x00,
00169         kI2cError                  = 0x01,
00170         kI2cTxPartial              = 0x02,
00171         kI2cRxOverflow             = 0x04,
00172         kI2cInProgress            = 0x06
00173     };
00174
00175
00176
00182     enum I2cPullups
00183     {
00184         kPullupsOff                = 0,
00185         kPullupsOn                 = 1
00186     };
00187
00188
00189
00190
00204     void start( uint8_t ownAddress, uint8_t speed = kI2cBusFast, bool answerGeneralCall = false );
00205
00206
00212     void stop();
00213
00214
00223     void pullups( uint8_t set = kPullupsOn );
00224
00225
00232     bool busy();
00233
00234
00235 #ifdef DEBUG_I2cSlaveDiary
00236
00237     void setDebugSout( Serial0* s );
00238
00239     void clearDebugI2cDiary();
00240     void dumpDebugI2cDiary();
00241
00242 #endif
00243
00244 };
00245
00246
00247 #endif
00248

```

10.19 InitSystem.h File Reference

Include this file to use the functions that initialize the microcontroller to a known, basic state.

Functions

- void [initSystem](#) ()

This function initializes the microcontroller by clearing any bootloader settings, clearing all timers, and turning on interrupts.

10.19.1 Detailed Description

Include this file to use the functions that initialize the microcontroller to a known, basic state.

To use these functions, include [InitSystem.h](#) in your source code and link against InitSystem.cpp.

10.19.2 Function Documentation

10.19.2.1 initSystem()

```
void initSystem ( )
```

This function initializes the microcontroller by clearing any bootloader settings, clearing all timers, and turning on interrupts.

This function is generally called at the very beginning of `main()`.

10.20 InitSystem.h

[Go to the documentation of this file.](#)

```
00001 /*
00002     SystemClock.cpp - Functions to initialize and use a system clock
00003     on AVR chips that is compatible with Arduino.
00004     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00005     This is part of the AVRTools library.
00006     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00007     Functions readlong() and readFloat() adapted from Arduino code that
00008     is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00009
00010     This program is free software: you can redistribute it and/or modify
00011     it under the terms of the GNU Lesser General Public License as published by
00012     the Free Software Foundation, either version 3 of the License, or
00013     (at your option) any later version.
00014
00015     This program is distributed in the hope that it will be useful,
00016     but WITHOUT ANY WARRANTY; without even the implied warranty of
00017     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00018     GNU Lesser General Public License for more details.
00019
00020     You should have received a copy of the GNU Lesser General Public License
00021     along with this program. If not, see <http://www.gnu.org/licenses/>.
00022 */
00023
00024
00037 #ifndef InitSystem_h
00038 #define InitSystem_h
00039
00040
00049 void initSystem();
00050
00051
00052 #endif
```

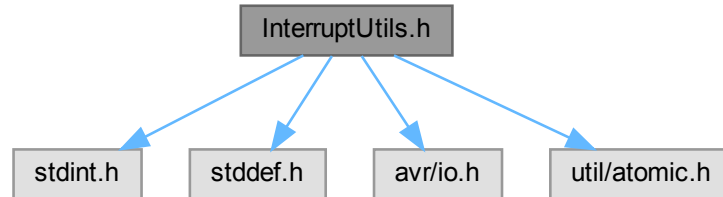
10.21 InterruptUtils.h File Reference

This file provides utilities for temporarily disabling (suppressing) interrupts of various kinds in a block of code. It uses the C++ RAII paradigm to ensure interrupt state is restored automatically when the block of code is exited. While all interrupts can be suppressed, tools are provided that allow more selective control of which interrupts are suppressed.

```
#include <stdint.h>
#include <stddef.h>
#include <avr/io.h>
```

```
#include <util/atomic.h>
```

Include dependency graph for InterruptUtils.h:



Classes

- class [Interrupts::AllOff](#)

This class defines an object that disables all interrupts during its lifetime. Interrupt state is restored by the object's destructor when the object goes out of scope.

- class [Interrupts::ExternalOff](#)

This class defines an object that disables selected external interrupts during its lifetime. The selected external interrupts are restored by the object's destructor when it goes out of scope.

- class [Interrupts::PinChangeOff](#)

This class defines an object that disables selected pin change interrupts during its lifetime. The selected pin change interrupts are restored by the object's destructor when it goes out of scope.

Namespaces

- namespace [Interrupts](#)

This namespace bundles various utility classes designed to suppress selected interrupts using the RAII idiom.

Enumerations

- enum [Interrupts::ExternalInterrupts](#) {
[Interrupts::kExternalInterrupt0](#) , [Interrupts::kExternalInterrupt1](#) , [Interrupts::kExternalInterrupt2](#) , [Interrupts::kExternalInterrupt3](#)
 ,
[Interrupts::kExternalInterrupt4](#) , [Interrupts::kExternalInterrupt5](#) , [Interrupts::kExternalInterrupt6](#) , [Interrupts::kExternalInterrupt7](#)
 ,
[Interrupts::kExternalInterruptAll](#) }

This enum lists the external interrupts that can be suppressed (disabled). To pass more than one external interrupt, simply "or" them.

- enum [Interrupts::PinChangeInterrupts](#) { [Interrupts::kPinChangeInterrupt0](#) , [Interrupts::kPinChangeInterrupt1](#) ,
[Interrupts::kPinChangeInterrupt2](#) , [Interrupts::kPinChangeInterruptAll](#) }

This enum lists the pin change interrupts that can be suppressed (disabled). To pass more than one pin change interrupt, simply "or" them.

10.21.1 Detailed Description

This file provides utilities for temporarily disabling (suppressing) interrupts of various kinds in a block of code. It uses the C++ RAII paradigm to ensure interrupt state is restored automatically when the block of code is exited. While all interrupts can be suppressed, tools are provided that allow more selective control of which interrupts are suppressed.

10.22 InterruptUtils.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      InterruptUtils.h - Utilities for managing interrupts for
00003      AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004      This is part of the AVRTools library.
00005      Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00006      Functions printNumber() and printFloat() adapted from Arduino code that
00007      is Copyright (c) 2008 David A. Mellis and licensed under LGPL.
00008
00009      This program is free software: you can redistribute it and/or modify
00010      it under the terms of the GNU Lesser General Public License as published by
00011      the Free Software Foundation, either version 3 of the License, or
00012      (at your option) any later version.
00013
00014      This program is distributed in the hope that it will be useful,
00015      but WITHOUT ANY WARRANTY; without even the implied warranty of
00016      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00017      GNU Lesser General Public License for more details.
00018
00019      You should have received a copy of the GNU Lesser General Public License
00020      along with this program. If not, see <http://www.gnu.org/licenses/>.
00021  */
00022
00023
00036  #ifndef InterruptUtils_h
00037  #define InterruptUtils_h
00038
00039  #include <stdint.h>
00040  #include <stddef.h>
00041
00042  #include <avr/io.h>
00043  #include <util/atomic.h>
00044
00045
00046
00047
00048
00054  namespace Interrupts
00055  {
00056
00063      class Alloff
00064      {
00065      public:
00066
00071          Alloff()
00072          {
00073              mSreg = SREG;
00074              cli();
00075          }
00076
00077
00083          ~Alloff()
00084          {
00085              // Turn on global interrupt, only if it was already on. Leave other bits alone.
00086              if ( mSreg & static_cast<uint8_t>(1 << SREG_I) )
00087              {
00088                  sei();
00089              }
00090          }
00091
00092
00093      private:
00094
00095          uint8_t mSreg;

```

```

00096     };
00097
00098
00099
00100
00101
00102
00103
00104 #if defined(__AVR_ATmega328P__)
00105 #define kExternalInterruptMask      0x03
00106 #elif defined(__AVR_ATmega2560__)
00107 #define kExternalInterruptMask      0xFF
00108 #else
00109 #error "Undefined AVR processor type"
00110 #endif
00111
00112
00120     enum ExternalInterrupts
00121     {
00122         kExternalInterrupt0      = ( 1 << INT0 ),
00123         kExternalInterrupt1      = ( 1 << INT1 ),
00124
00125 #if defined(__AVR_ATmega2560__)
00126         kExternalInterrupt2      = ( 1 << INT2 ),
00127         kExternalInterrupt3      = ( 1 << INT3 ),
00128         kExternalInterrupt4      = ( 1 << INT4 ),
00129         kExternalInterrupt5      = ( 1 << INT5 ),
00130         kExternalInterrupt6      = ( 1 << INT6 ),
00131         kExternalInterrupt7      = ( 1 << INT7 ),
00132 #endif
00133
00134         kExternalInterruptAll     = kExternalInterruptMask
00135     };
00136
00137
00138
00144     class ExternalOff
00145     {
00146     public:
00147
00157         ExternalOff( uint8_t whichOnesToTurnOff = kExternalInterruptMask )
00158         : mExternalInterruptsToSuppress( whichOnesToTurnOff & kExternalInterruptMask )
00159         {
00160             // Disable the selected interrupts
00161             EIMSK &= ~(mExternalInterruptsToSuppress);
00162         }
00163
00164
00170         ~ExternalOff()
00171         {
00172             // Enable the selected interrupts
00173             EIMSK |= mExternalInterruptsToSuppress;
00174         }
00175
00176
00177     private:
00178
00179         uint8_t mExternalInterruptsToSuppress;
00180     };
00181
00182
00183
00184
00185
00186
00187
00188 #if defined(__AVR_ATmega328P__)
00189 #define kPinChangeInterruptMask      0x07
00190 #elif defined(__AVR_ATmega2560__)
00191 #define kPinChangeInterruptMask      0x07
00192 #else
00193 #error "Undefined AVR processor type"
00194 #endif
00195
00196
00204     enum PinChangeInterrupts
00205     {
00206         kPinChangeInterrupt0      = ( 1 << PCINT0 ),
00207         kPinChangeInterrupt1      = ( 1 << PCINT1 ),
00208         kPinChangeInterrupt2      = ( 1 << PCINT2 ),
00209         kPinChangeInterruptAll     = kPinChangeInterruptMask

```

```

00210     };
00211
00212
00213
00219     class PinChangeOff
00220     {
00221     public:
00222
00231         PinChangeOff( uint8_t whichOnesToTurnOff = kPinChangeInterruptMask )
00232         : mPinChangeInterruptsToSuppress( whichOnesToTurnOff & kPinChangeInterruptMask )
00233         {
00234             // Disable the selected interrupts
00235             PCICR &= ~(mPinChangeInterruptsToSuppress);
00236         }
00237
00238
00243         ~PinChangeOff()
00244         {
00245             // Enable the selected interrupts
00246             PCICR |= mPinChangeInterruptsToSuppress;
00247         }
00248
00249
00250     private:
00251
00252         uint8_t mPinChangeInterruptsToSuppress;
00253     };
00254
00255
00256
00257 }; // End namespace
00258
00259
00260 #endif
00261

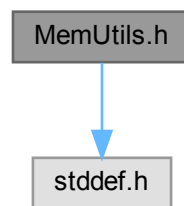
```

10.23 MemUtils.h File Reference

This file provides functions that provide information on the available memory in SRAM.

```
#include <stddef.h>
```

Include dependency graph for MemUtils.h:



Namespaces

- namespace [MemUtils](#)

A namespace providing encapsulation for functions that report the available memory in SRAM.

Functions

- `size_t MemUtils::freeSRAM ()`
Get the total free memory remaining in SRAM.
- `size_t MemUtils::freeMemoryBetweenHeapAndStack ()`
Get the free memory between the heap and the stack.
- `void MemUtils::resetHeap ()`
Reset the heap to an empty (virgin) state.
- `size_t MemUtils::memoryAvailableOnFreeList ()`
Get the free memory on the heap free-list.
- `size_t MemUtils::getFreeListStats (int *nbrBlocks, size_t *sizeSmallestBlock, size_t *sizeLargestBlock)`
Get information about the heap free-list.

10.23.1 Detailed Description

This file provides functions that provide information on the available memory in SRAM.

To use these functions, include [MemUtils.h](#) in your source code and link against `MemUtils.cpp`.

These functions are wrapped in namespace [MemUtils](#) to avoid namespace collisions.

10.24 MemUtils.h

[Go to the documentation of this file.](#)

```
00001 /*
00002     MemUtils.h - Memory-related utilities
00003     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007     This program is free software: you can redistribute it and/or modify
00008     it under the terms of the GNU General Public License as published by
00009     the Free Software Foundation, either version 3 of the License, or
00010     (at your option) any later version.
00011
00012     This program is distributed in the hope that it will be useful,
00013     but WITHOUT ANY WARRANTY; without even the implied warranty of
00014     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015     GNU General Public License for more details.
00016
00017     You should have received a copy of the GNU General Public License
00018     along with this program. If not, see <http://www.gnu.org/licenses/>.
00019 */
00020
00021
00022
00036 #ifndef MemUtils_h
00037 #define MemUtils_h
00038
00039 #include <stddef.h>
00040
00045 namespace MemUtils
00046 {
00047
00059     size_t freeSRAM();
00060
00061
00062
00073     size_t freeMemoryBetweenHeapAndStack();
00074 }
```



```
00075
00076
00086     void resetHeap();
00087
00088
00089
00103     size_t memoryAvailableOnFreeList();
00104
00105
00106
00124     size_t getFreeListStats( int* nbrBlocks, size_t* sizeSmallestBlock, size_t* sizeLargestBlock );
00125
00126 };
00127
00128 #endif
```

10.25 new.h File Reference

This file provides `operator new` and `operator delete`. You only need this file if you use `new` and `delete` to manage objects on the heap.

```
#include <stdlib.h>
```

Include dependency graph for new.h:



10.25.1 Detailed Description

This file provides `operator new` and `operator delete`. You only need this file if you use `new` and `delete` to manage objects on the heap.

If you do use `new` and `delete`, then include [new.h](#) in your source files and link your project against `new.cpp`.

Note

The AVRTools library does not itself make any use of heap storage or the `new` or `delete` operators.

10.26 new.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      new.cpp - operator new implementations not provided with avr-gcc.
00003      This is part of the AVRTools library.
00004      Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00005
00006      This program is free software: you can redistribute it and/or modify
00007      it under the terms of the GNU General Public License as published by
00008      the Free Software Foundation, either version 3 of the License, or
00009      (at your option) any later version.
00010
00011      This program is distributed in the hope that it will be useful,
00012      but WITHOUT ANY WARRANTY; without even the implied warranty of
00013      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00014      GNU General Public License for more details.
00015
00016      You should have received a copy of the GNU General Public License
00017      along with this program. If not, see <http://www.gnu.org/licenses/>.
00018  */
00019
00020
00021
00037  #ifndef new_h
00038  #define new_h
00039
00040
00041  #include <stdlib.h>
00042
00043  void* operator new( size_t size );
00044  void* operator new[]( size_t size );
00045
00046
00047  void operator delete( void* ptr );
00048  void operator delete[]( void* ptr );
00049
00050  #if __cplusplus >= 201402L
00051
00052  void operator delete ( void* ptr, size_t sz );
00053  void operator delete[]( void* ptr, size_t sz );
00054
00055  #endif
00056
00057
00058  // Placement new & delete operators
00059
00060  inline void* operator new( size_t, void* ptr )
00061  { return ptr; }
00062  inline void* operator new[]( size_t, void* ptr )
00063  { return ptr; }
00064
00065  inline void operator delete( void* , void* )
00066  { }
00067  inline void operator delete[]( void* , void* )
00068  { }
00069
00070
00071  #endif
00072

```

10.27 Pwm.h File Reference

This file provides functions that access the PWM capability of the ATmega328 and ATmega2560 microcontrollers.

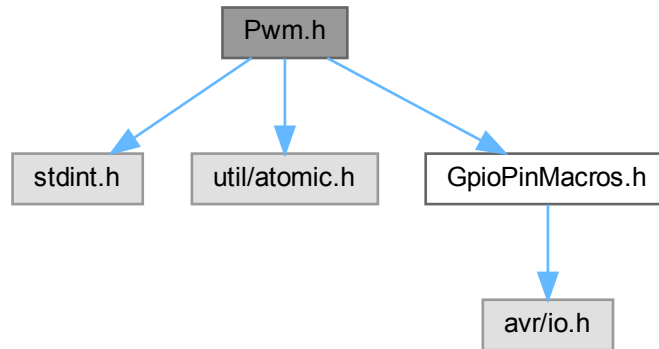
```

#include <stdint.h>
#include <util/atomic.h>

```

```
#include "GpioPinMacros.h"
```

Include dependency graph for Pwm.h:



Macros

- #define [writeGpioPinPwm](#)(pinName, value)
Write a PWM value to a pin.

Functions

- void [writeGpioPinPwmV](#) (const [GpioPinVariable](#) &pinVar, uint8_t value)
Write a PWM value to a pin.
- void [initPwmTimer0](#) ()
Initialize timer0 for PWM.
- void [initPwmTimer1](#) ()
Initialize timer1 for PWM.
- void [initPwmTimer2](#) ()
Initialize timer2 for PWM.
- void [clearTimer0](#) ()
Clear timer0.
- void [clearTimer1](#) ()
Clear timer1.
- void [clearTimer2](#) ()
Clear timer2.
- void [initPwmTimer3](#) ()
Initialize timer3 for PWM.
- void [initPwmTimer4](#) ()
Initialize timer4 for PWM.
- void [initPwmTimer5](#) ()

Initialize timer5 for PWM.

- void `clearTimer3()`

Clear timer3.

- void `clearTimer4()`

Clear timer4.

- void `clearTimer5()`

Clear timer5.

10.27.1 Detailed Description

This file provides functions that access the PWM capability of the ATmega328 and ATmega2560 microcontrollers.

To use these functions, include `Pwm.h` in your source code and link against `Pwm.cpp`.

Before you use the `writePinPwm()` function, you must first initialize the appropriate timers using the appropriate `initPwmTimerN()` function.

The association between PWN pins and timers is as follows:

For Arduino Uno (ATmega328)

Arduino Uno pin	ATmega328 pin	Timer
3	PD3	timer2
5	PD5	timer0
6	PD6	timer0
9	PB1	timer1
10	PB2	timer1
11	PB3	timer2

For Arduino Mega (ATmega2560)

Arduino Mega pin	ATmega2560 pin	Timer
2	PE4	timer3
3	PE5	timer3
4	PG5	timer0
5	PE3	timer3
6	PH3	timer4
7	PH4	timer4
8	PH5	timer4
9	PH6	timer2
10	PB4	timer2
11	PB5	timer1
12	PB6	timer1
13	PB7	timer0
44	PL5	timer5
45	PL4	timer5
46	PL3	timer5

Note

Timer0 is also used by the system clock. *Do not initialize or clear timer0* if you are also using the system clock function from [SystemClock.h](#). If you are using the system clock function, you can use timer0-based PWM functions *without* having to call [initPwmTimer0\(\)](#).

10.27.2 Macro Definition Documentation

10.27.2.1 writeGpioPinPwm

```
#define writeGpioPinPwm(  
    pinName,  
    value )
```

Write a PWM value to a pin.

This sets the duty cycle for the PWM on the pin. Completely off is represented by 0; completely on is represented by 1.

Before calling this function, you must initialize the appropriate timer by calling [initPwmTimerN\(\)](#), where N = 1, 2, 3, 4, or 5 is the timer corresponding to that particular pin.

- `pinName` a pin name macro generated by [GpioPinPwm\(\)](#).
- `value` a value between 0 and 255.

Warning

Timer0 is also used by the system clock. *Do not initialize or clear timer0* if you are also using the system clock function from [SystemClock.h](#). If you are using the system clock function, you can use timer0-based PWM functions *without* having to call [initPwmTimer0\(\)](#).

Note

You can temporarily turn off PWM by writing a 0 to the pin with `writePinPwm(pin, 0)`. In particular, this is how to turn off PWM to pins associated with timer0 when timer0 is also being used by the system clock.

This macro ensures operations on 16-bit timers are atomic (at the cost of a small amount of overhead in the case of 8-bit timers).

10.27.3 Function Documentation

10.27.3.1 clearTimer0()

```
void clearTimer0 ( )
```

Clear timer0.

This function clears timer0.

Note

Timer0 is also used by the system clock. *Do not clear timer0* if you are also using the system clock function from [SystemClock.h](#).

Only call this function if you called [initPwmTimer0\(\)](#) instead of [initSystemClock\(\)](#).

Note

To turn off PWM on pins associated with timer0 while also using the system clock, write a zero to the pin by calling `writePinPwm(pinName, 0)`.

10.27.3.2 clearTimer1()

```
void clearTimer1 ( )
```

Clear timer1.

This function clears timer1, turning off the PWM functionality.

10.27.3.3 clearTimer2()

```
void clearTimer2 ( )
```

Clear timer2.

This function clears timer2, turning off the PWM functionality.

10.27.3.4 clearTimer3()

```
void clearTimer3 ( )
```

Clear timer3.

This function clears timer3, turning off the PWM functionality.

Note

This function is only available on Arduino Mega (ATmega2560).

10.27.3.5 clearTimer4()

```
void clearTimer4 ( )
```

Clear timer4.

This function clears timer4, turning off the PWM functionality.

Note

This function is only available on Arduino Mega (ATmega2560).

10.27.3.6 clearTimer5()

```
void clearTimer5 ( )
```

Clear timer5.

This function clears timer5, turning off the PWM functionality.

Note

This function is only available on Arduino Mega (ATmega2560).

10.27.3.7 initPwmTimer0()

```
void initPwmTimer0 ( )
```

Initialize timer0 for PWM.

This function sets timer0 for phase-correct PWM mode. You must call this function or `initSystemClock()` before calling `writePinPwm()` on a PWM pin associated with timer0.

The PWM pins supported by timer0 are:

- Arduino Uno (ATmega328): pin 5 (PD5), pin 6 (PD6)
- Arduino Mega (ATmega2560): pin 4 (PG5), pin 13 (PB7)

Note

Timer0 is also used by the system clock. *Do not initialize timer0* if you are also using the system clock function from [SystemClock.h](#).

The function `initSystemClock()` puts timer0 in fast PWM mode. While this is different than the phase-correct PWM mode preferred for PWM usage, fast PWM mode still allows PWM operations on the associated pins. However, the duty cycles may be slightly off, and calling `writePinPwm(pin, 0)` may not completely turn off output on the pins associated with timer0.

Only call `initPwmTimer0()` if you did *not* call `initSystemClock()` (i.e., you are *not* using the system clock) and you wish to use PWM on the pins associate with timer0.

Note

To turn off PWM on pins associated with timer0 while also using the system clock, write a zero to the pin by calling `writePinPwm(pinName, 0)`.

10.27.3.8 initPwmTimer1()

```
void initPwmTimer1 ( )
```

Initialize timer1 for PWM.

This function sets timer1 for phase-correct PWM mode. You must call this function before calling writePinPwm() on a PWM pin associated with timer1.

The PWM pins supported by timer1 are:

- Arduino Uno (ATmega328): pin 9 (PB1), pin 10 (PB2)
- Arduino Mega (ATmega2560): pin 11 (PB5), pin 12 (PB6)

10.27.3.9 initPwmTimer2()

```
void initPwmTimer2 ( )
```

Initialize timer2 for PWM.

This function sets timer2 for phase-correct PWM mode. You must call this function before calling writePinPwm() on a PWM pin associated with timer2.

The PWM pins supported by timer2 are:

- Arduino Uno (ATmega328): pin 3 (PD3), pin 11 (PB3)
- Arduino Mega (ATmega2560): pin 9 (PH6), pin 10 (PB4)

10.27.3.10 initPwmTimer3()

```
void initPwmTimer3 ( )
```

Initialize timer3 for PWM.

This function sets timer3 for phase-correct PWM mode. You must call this function before calling writePinPwm() on a PWM pin associated with timer3.

The PWM pins supported by timer3 are:

- Arduino Mega (ATmega2560): pin 2 (PE4), pin 3 (PE5)

Note

This function is only available on Arduino Mega (ATmega2560).

10.27.3.11 initPwmTimer4()

```
void initPwmTimer4 ( )
```

Initialize timer4 for PWM.

This function sets timer4 for phase-correct PWM mode. You must call this function before calling writePinPwm() on a PWM pin associated with timer4.

The PWM pins supported by timer4 are:

- Arduino Mega (ATmega2560): pin 6 (PH3), pin 7 (PH4), pin 8 (PH5)

Note

This function is only available on Arduino Mega (ATmega2560).

10.27.3.12 initPwmTimer5()

```
void initPwmTimer5 ( )
```

Initialize timer5 for PWM.

This function sets timer5 for phase-correct PWM mode. You must call this function before calling writePinPwm() on a PWM pin associated with timer5.

The PWM pins supported by timer5 are:

- Arduino Mega (ATmega2560): pin 44 (PL5), pin 45 (PL4), pin 46 (PL3)

Note

This function is only available on Arduino Mega (ATmega2560).

10.27.3.13 writeGpioPinPwmV()

```
void writeGpioPinPwmV (
    const GpioPinVariable & pinVar,
    uint8_t value ) [inline]
```

Write a PWM value to a pin.

This sets the duty cycle for the PWM on the pin. Completely off is represented by 0; completely on is represented by 1.

Before calling this function, you must initialize the appropriate timer by calling `initPwmTimerN()`, where N = 1, 2, 3, 4, or 5 is the timer corresponding to that particular pin.

- `pinVar` a pin variable that has PWM capabilities (i.e., initialized with `makeGpioVarFromGpioPinPwm()`).
- `value` a value between 0 and 255.

Warning

Timer0 is also used by the system clock. *Do not initialize or clear timer0* if you are also using the system clock function from `SystemClock.h`. If you are using the system clock function, you can use timer0-based PWM functions *without* having to call `initPwmTimer0()`.

Note

You can temporarily turn off PWM by writing a 0 to the pin with `writePinPwm(pin, 0)`. In particular, this is how to turn off PWM to pins associated with timer0 when timer0 is also being used by the system clock.

This function ensures operations on 16-bit timers are atomic (at the cost of a small amount of overhead in the case of 8-bit timers).

10.28 Pwm.h

[Go to the documentation of this file.](#)

```
00001 /*
00002  Pwm.h - Macros and Functions for accessing the PMW capabilities of AVR's.
00003  For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004  This is part of the AVRTools library.
00005  Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007  This program is free software: you can redistribute it and/or modify
00008  it under the terms of the GNU General Public License as published by
00009  the Free Software Foundation, either version 3 of the License, or
00010  (at your option) any later version.
00011
00012  This program is distributed in the hope that it will be useful,
00013  but WITHOUT ANY WARRANTY; without even the implied warranty of
00014  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015  GNU General Public License for more details.
00016
00017  You should have received a copy of the GNU General Public License
00018  along with this program. If not, see <http://www.gnu.org/licenses/>.
00019 */
00020
00021
00022
```

```

00076 #ifndef Pwm_h
00077 #define Pwm_h
00078
00079
00080 #include <stdint.h>
00081
00082 #include <util/atomic.h>
00083
00084 #include "GpioPinMacros.h"
00085
00086
00087
00088 #define _writeGpioPinPwm( ddr, port, pin, nbr, chl, ocr, com, tccr, value )
00089 do
00090 {
00091     if ( value <= 0 )
00092     {
00093         tccr &= ~(1<<com);
00094         port &= ~(1<<nbr);
00095     }
00096     else if ( value >= 255 )
00097     {
00098         tccr &= ~(1<<com);
00099         port |= (1<<nbr);
00100     }
00101     else
00102     {
00103         tccr |= (1<<com);
00104         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00105         {
00106             ocr = value;
00107         }
00108     }
00109 }
00110 while ( 0 )
00111
00112
00113
00143 #define writeGpioPinPwm( pinName, value )      _writeGpioPinPwm( pinName, value )
00144
00145
00146
00147
00176 inline void writeGpioPinPwmV( const GpioPinVariable& pinVar, uint8_t value )
00177 {
00178     if ( value == 0 )
00179     {
00180         *(pinVar.tccr()) &= ~( 1 << pinVar.com() );
00181         *(pinVar.port()) &= ~( 1 << pinVar.bitNbr() );
00182     }
00183     else if ( value == 255 )
00184     {
00185         *(pinVar.tccr()) &= ~( 1 << pinVar.com() );
00186         *(pinVar.port()) |= ( 1 << pinVar.bitNbr() );
00187     }
00188     else
00189     {
00190         *(pinVar.tccr()) |= ( 1 << pinVar.com() );
00191         // Provide atomicity for 16-bit timers (not needed for 8-bit timers, but be safe)
00192         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00193         {
00194             *(pinVar.ocr()) = value;
00195         }
00196     }
00197 }
00198 }
00199
00200
00201
00229 void initPwmTimer0();
00230
00231
00244 void initPwmTimer1();
00245
00246
00259 void initPwmTimer2();
00260
00261
00276 void clearTimer0();
00277
00278

```

```
00279
00280
00287 void clearTimer1();
00288
00289
00296 void clearTimer2();
00297
00298
00299 #if defined(__AVR_ATmega2560__)
00300
00301
00315 void initPwmTimer3();
00316
00317
00331 void initPwmTimer4();
00332
00333
00334
00348 void initPwmTimer5();
00349
00350
00351
00352
00361 void clearTimer3();
00362
00363
00372 void clearTimer4();
00373
00374
00383 void clearTimer5();
00384
00385 #endif
00386
00387 #endif
00388
```

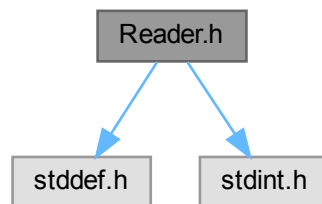
10.29 Reader.h File Reference

This file provides a generic interface to incoming data streams of any kind. It is designed around how serial streams are generally used, but can be used with any system that provides a sequential input of bytes that can be interpreted as strings and/or numbers.

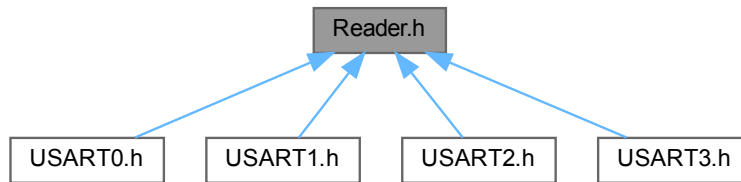
```
#include <stddef.h>
```

```
#include <stdint.h>
```

Include dependency graph for Reader.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [Reader](#)

This is an abstract class defining a generic interface to read numbers and strings from a sequential stream of bytes (such as a serial device).

10.29.1 Detailed Description

This file provides a generic interface to incoming data streams of any kind. It is designed around how serial streams are generally used, but can be used with any system that provides a sequential input of bytes that can be interpreted as strings and/or numbers.

10.30 Reader.h

[Go to the documentation of this file.](#)

```

00001 /*
00002     Reader.cpp - a base class for reading data
00003     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006     Functions readLong() and readFloat() adapted from Arduino code that
00007     is Copyright (c) 2008 David A. Mellis and licensed under LGPL.
00008
00009     This program is free software: you can redistribute it and/or modify
00010     it under the terms of the GNU Lesser General Public License as published by
00011     the Free Software Foundation, either version 3 of the License, or
00012     (at your option) any later version.
00013
00014     This program is distributed in the hope that it will be useful,
00015     but WITHOUT ANY WARRANTY; without even the implied warranty of
00016     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00017     GNU Lesser General Public License for more details.
00018
00019     You should have received a copy of the GNU Lesser General Public License
00020     along with this program. If not, see <http://www.gnu.org/licenses/>.
00021 */
00022
00032 #ifndef Reader_h
00033 #define Reader_h
00034
00035
00036 #include <stdint.h>
  
```

```
00037 #include <stdint.h>
00038
00039
00040 #ifndef SERIAL_INPUT_EOL
00041 #define SERIAL_INPUT_EOL    '\n'
00042 #endif
00043
00044
00065 class Reader
00066 {
00067
00068 public:
00069
00070
00075     Reader();
00076
00077
00078
00079     // Virtual methods (pure -- need to be implemented by derived classes)
00080
00087     virtual int read() = 0;
00088
00089
00096     virtual int peek() = 0;
00097
00098
00105     virtual bool available() = 0;
00106
00107
00108
00109
00110     // Parsing methods
00111
00117     void setTimeout( unsigned long milliseconds )
00118     { mTimeOut = milliseconds; }
00119
00120
00128     bool find( const char *target )
00129     { return findUntil( target, 0 ); }
00130
00131
00141     bool find( const char *target, size_t length )
00142     { return findUntil( target, length, NULL, 0 ); }
00143
00144
00157     bool findUntil( const char *target, const char *terminator );
00158
00159
00174     bool findUntil( const char *target, size_t targetLen, const char *terminate, size_t termLen );
00175
00176
00187     bool readLong( long* result );
00188
00189
00200     bool readFloat( float* result );
00201
00202
00217     bool readLong( long* result, char skipChar );
00218
00219
00234     bool readFloat( float* result, char skipChar );
00235
00236
00247     size_t readBytes( char *buffer, size_t length );
00248
00249
00262     size_t readBytesUntil( char terminator, char* buffer, size_t length );
00263
00264
00275     size_t readBytes( uint8_t* buffer, size_t length )
00276     { return readBytes( reinterpret_cast<char*>(buffer), length ); }
00277
00278
00279
00292     size_t readBytesUntil( uint8_t terminator, uint8_t* buffer, size_t length )
00293     { return readBytesUntil( static_cast<char>(terminator), reinterpret_cast<char*>(buffer), length ); }
00294
00295
00307     size_t readLine( char *buffer, size_t length );
00308
00309
```

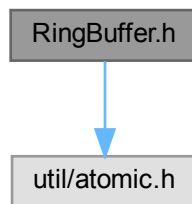
```
00314     void consumeWhiteSpace();
00315
00316
00317 private:
00318
00319     // Number of milliseconds to wait for the next char before aborting timed read
00320     unsigned long mTimeOut;
00321
00322     int timedRead();           // private method to read stream with timeout
00323     int timedPeek();          // private method to peek stream with timeout
00324     int peekNextDigit();      // returns the next numeric digit in the stream or -1 if timeout
00325 };
00326
00327
00328 #endif
```

10.31 RingBuffer.h File Reference

This file provides an efficient ring buffer implementation for storing bytes.

```
#include <util/atomic.h>
```

Include dependency graph for RingBuffer.h:



Classes

- class [RingBuffer](#)

This class provides an efficient ring buffer implementation for storing bytes. Ring buffers are particularly useful for memory constrained microcontrollers such as the ATmega328 and ATmega2650. For maximum efficiency, this class is focused on the storage of bytes, providing a single code base that is shared by all instances of this class.

10.31.1 Detailed Description

This file provides an efficient ring buffer implementation for storing bytes.

Ring buffers are particularly useful for memory constrained microcontrollers such as the ATmega328 and ATmega2650.

10.32 RingBuffer.h

[Go to the documentation of this file.](#)

```

00001  /*
00002     RingBuffer.h - A ring buffer class for AVR processors.
00003     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007     This program is free software: you can redistribute it and/or modify
00008     it under the terms of the GNU General Public License as published by
00009     the Free Software Foundation, either version 3 of the License, or
00010     (at your option) any later version.
00011
00012     This program is distributed in the hope that it will be useful,
00013     but WITHOUT ANY WARRANTY; without even the implied warranty of
00014     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015     GNU General Public License for more details.
00016
00017     You should have received a copy of the GNU General Public License
00018     along with this program. If not, see <http://www.gnu.org/licenses/>.
00019  */
00020
00021
00022
00034  #ifndef RingBuffer_h
00035  #define RingBuffer_h
00036
00037
00038  #include <util/atomic.h>
00039
00040
00041
00060  class RingBuffer
00061  {
00062  public:
00063
00070      RingBuffer( unsigned char *buffer, unsigned short size );
00071
00077      int pull();
00078
00087      int peek( unsigned short index = 0 );
00088
00097      bool push( unsigned char element );
00098
00099
00105      bool isFull();
00106
00112      bool isNotFull();
00113
00119      bool isEmpty()
00120      { return !static_cast<bool>( mLength ); }
00121
00127      bool isNotEmpty()
00128      { return static_cast<bool>( mLength ); }
00129
00133      void clear();
00134
00135
00136  private:
00137
00138      unsigned char *mBuffer;
00139      volatile unsigned short mSize;
00140      volatile unsigned short mLength;
00141      volatile unsigned short mIndex;
00142
00143  };
00144
00145
00146  #endif
00147
00148

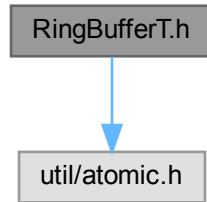
```


10.33 RingBufferT.h File Reference

This file provides a very flexible, template-based ring buffer implementation.

```
#include <util/atomic.h>
```

Include dependency graph for RingBufferT.h:



Classes

- class [RingBufferT< T, N, SIZE >](#)

a template-based ring buffer class that can store different kinds of objects in buffers of whatever size is needed.

10.33.1 Detailed Description

This file provides a very flexible, template-based ring buffer implementation.

Ring buffers are versatile storage structures. This file provides a template-based ring buffer implementation that can store different kinds of objects in buffers of whatever size is needed.

10.34 RingBufferT.h

[Go to the documentation of this file.](#)

```

00001 /*
00002  RingBufferT.h - A ring buffer template class for AVR processors.
00003  For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004  This is part of the AVRTools library.
00005  Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006
00007  This program is free software: you can redistribute it and/or modify
00008  it under the terms of the GNU General Public License as published by
00009  the Free Software Foundation, either version 3 of the License, or
00010  (at your option) any later version.
00011
00012  This program is distributed in the hope that it will be useful,
00013  but WITHOUT ANY WARRANTY; without even the implied warranty of
00014  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00015  GNU General Public License for more details.
00016

```

```

00017     You should have received a copy of the GNU General Public License
00018     along with this program.  If not, see <http://www.gnu.org/licenses/>.
00019 */
00020
00021
00022
00023
00035 #ifndef RingBufferT_h
00036 #define RingBufferT_h
00037
00038
00039 #include <util/atomic.h>
00040
00041
00060 template< typename T, typename N, unsigned int SIZE > class RingBufferT
00061 {
00062
00063 public:
00064
00069     RingBufferT()
00070         : mSize( SIZE ), mLength( 0 ), mIndex( 0 )
00071     {}
00072
00073
00083     T pull()
00084     {
00085         T element = 0;
00086         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00087         {
00088             if ( mLength )
00089             {
00090                 element = mBuffer[ mIndex ];
00091                 mIndex++;
00092                 if ( mIndex >= mSize )
00093                 {
00094                     mIndex -= mSize;
00095                 }
00096                 --mLength;
00097             }
00098         }
00099         return element;
00100     }
00101
00102
00115     T peek( N index = 0 )
00116     {
00117         T element;
00118         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00119         {
00120             element = mBuffer[ ( mIndex + index ) % mSize ];
00121         }
00122         return element;
00123     }
00124
00125
00134     bool push( T element )
00135     {
00136         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00137         {
00138             if ( mLength < mSize )
00139             {
00140                 mBuffer[ ( mIndex + mLength ) % mSize ] = element;
00141                 ++mLength;
00142                 return 0;
00143             }
00144             // True = failure
00145             return 1;
00146         }
00147     }
00148
00149
00155     bool isEmpty()
00156     {
00157         return !static_cast<bool>( mLength );
00158     }
00159
00160
00166     bool isEmpty()
00167     {
00168         return static_cast<bool>( mLength );
00169     }

```

```

00170
00171
00177     bool isFull()
00178     {
00179         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00180         {
00181             return ( mSize - mLength ) <= 0;
00182         }
00183     }
00184
00185
00191     bool isNotFull()
00192     {
00193         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00194         {
00195             return ( mSize - mLength ) > 0;
00196         }
00197     }
00198
00199
00205     void discardFromFront( N nbrElements )
00206     {
00207         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00208         {
00209             if ( nbrElements < mLength )
00210             {
00211                 mIndex += nbrElements;
00212                 if( mIndex >= mSize )
00213                 {
00214                     mIndex -= mSize;
00215                 }
00216                 mLength -= nbrElements;
00217             }
00218             else
00219             {
00220                 // flush the whole buffer
00221                 mLength = 0;
00222             }
00223         }
00224     }
00225
00226
00230     void clear()
00231     {
00232         ATOMIC_BLOCK( ATOMIC_RESTORESTATE )
00233         {
00234             mLength = 0;
00235         }
00236     }
00237
00238
00239
00240 private:
00241
00242     T mBuffer[ SIZE ] ;
00243     volatile N mSize;
00244     volatile N mLength;
00245     volatile N mIndex;
00246
00247 };
00248
00249
00250 #endif
00251
00252

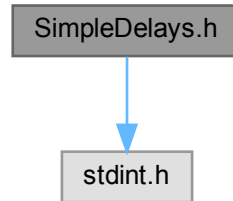
```

10.35 SimpleDelays.h File Reference

This file provides simple delay functions that do not involve timers or interrupts. These functions simply execute a series of nested loops with known and precise timing.

```
#include <stdint.h>
```

Include dependency graph for SimpleDelays.h:



Functions

- void [delayQuartersOfMicroSeconds](#) (uint16_t nbrOfQuartersOfMicroSeconds)
Delay a given number of quarter microseconds. Due to function call overhead, at 16 MHz the smallest possible delay is just under 6 quarter microseconds (~1.5 microseconds). Delays of 7 quarter microseconds or greater are reasonably accurate. At 8 MHz the smallest possible delay is just under 12 quarter microseconds (~3 microseconds). Delays of 13 quarter microseconds or greater are reasonably accurate.
- void [delayWholeMilliSeconds](#) (uint8_t nbrOfMilliSeconds)
Delay a given number of milliseconds. Despite function call overhead, this function is accurate within a few hundreds of microseconds.
- void [delayTenthsOfSeconds](#) (uint8_t nbrOfTenthsOfSeconds)
Delay a given number of tenths of a seconds. Despite function call overhead, this function is accurate within a few hundreds of microseconds.

10.35.1 Detailed Description

This file provides simple delay functions that do not involve timers or interrupts. These functions simply execute a series of nested loops with known and precise timing.

For precision, these functions are all implemented directly in assembler.

Note

These functions are implemented for (and automatically adjust to) either an 8 MHz, 12 MHz, or a 16 MHz clock cycle.

10.35.2 Function Documentation

10.35.2.1 delayQuartersOfMicroSeconds()

```
void delayQuartersOfMicroSeconds (
    uint16_t nbrOfQuartersOfMicroSeconds )
```

Delay a given number of quarter microseconds. Due to function call overhead, at 16 MHz the smallest possible delay is just under 6 quarter microseconds (~ 1.5 microseconds). Delays of 7 quarter microseconds or greater are reasonably accurate. At 8 MHz the smallest possible delay is just under 12 quarter microseconds (~ 3 microseconds). Delays of 13 quarter microseconds or greater are reasonably accurate.

At 16 MHz delays of less than 7 quarter microseconds produce a delay of just under 6 quarter microseconds (~ 1.5 microseconds). At 8 MHz delays of less than 12 quarter microseconds produce a delay of just under 12 quarter microseconds (~ 3 microseconds).

The maximum delay is 65535 quarter microseconds (equal to 16,383.75 microseconds, or about 16.4 milliseconds).

- `nbrOfQuartersOfMicroSeconds` the number of quarter microseconds to delay. For 16 MHz clocks, arguments less than 7 quarter microseconds for 16 MHz clocks all produce delays of just under 6 quarter microseconds. For 8 MHz clocks, arguments less than 13 quarter microseconds all produce delays of about 12 quarter microseconds.

Note

This delay function is only accurate if interrupts are disabled. If interrupts are enabled, the delays will be at least as long as requested, but may actually be longer. If accurate delays are desired, disable interrupts before calling this function (remember to enable interrupts afterwards).

This function only works for CPU clocks running at either 8 MHz, 12 MHz, or 16 MHz.

For precision, this function is implemented directly in assembler.

10.35.2.2 delayTenthsOfSeconds()

```
void delayTenthsOfSeconds (
    uint8_t nbrOfTenthsOfSeconds )
```

Delay a given number of tenths of a seconds. Despite function call overhead, this function is accurate within a few hundreds of microseconds.

- `nbrOfTenthsOfSeconds` the number of tenths of seconds to delay. The maximum delay is 256 tenths of a second or 25.6 seconds (pass 0 for a delay of 256 tenths of a second).

Note

This delay function is only accurate if interrupts are disabled. If interrupts are enabled, the delays will be at least as long as requested, but may actually be longer. If accurate delays are desired, disable interrupts before calling this function (remember to enable interrupts afterwards).

This function only works for CPU clocks running at either 8 MHz, 12 MHz, or 16 MHz.

For precision, this function is implemented directly in assembler.

10.35.2.3 delayWholeMilliseconds()

```
void delayWholeMilliseconds (
    uint8_t nbrOfMilliseconds )
```

Delay a given number of milliseconds. Despite function call overhead, this function is accurate within a few hundreds of microseconds.

- `nbrOfMilliseconds` the number of milliseconds to delay. The maximum delay is 256 milliseconds (pass 0 for a delay of 256 milliseconds).

Note

This delay function is only accurate if interrupts are disabled. If interrupts are enabled, the delays will be at least as long as requested, but may actually be longer. If accurate delays are desired, disable interrupts before calling this function (remember to enable interrupts afterwards).

This function only works for CPU clocks running at either 8 MHz, 12 MHz, or 16 MHz.

For precision, this function is implemented directly in assembler.

10.36 SimpleDelays.h

[Go to the documentation of this file.](#)

```
00001 /*
00002     SimpleDelays.h - Simple delay functions.
00003     This is part of the AVRTools library.
00004     Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00005
00006     This program is free software: you can redistribute it and/or modify
00007     it under the terms of the GNU General Public License as published by
00008     the Free Software Foundation, either version 3 of the License, or
00009     (at your option) any later version.
00010
00011     This program is distributed in the hope that it will be useful,
00012     but WITHOUT ANY WARRANTY; without even the implied warranty of
00013     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00014     GNU General Public License for more details.
00015
00016     You should have received a copy of the GNU General Public License
00017     along with this program. If not, see <http://www.gnu.org/licenses/>.
00018 */
00019
00020
00021
00037 #ifndef SimpleDelays_h
00038 #define SimpleDelays_h
00039
00040
00041 #include <stdint.h>
00042
00043
00044 #ifdef __cplusplus
00045 extern "C" {
00046 #endif
00047
00048
00049
00050
00051
00052
00083 void delayQuartersOfMicroSeconds( uint16_t nbrOfQuartersOfMicroSeconds );
```

```

00084
00085
00086
00087
00088
00107 void delayWholeMilliseconds( uint8_t nbrOfMilliseconds );
00108
00109
00110
00111
00112
00131 void delayTenthsOfSeconds( uint8_t nbrOfTenthsOfSeconds );
00132
00133
00134
00135
00136
00137 #ifdef __cplusplus
00138 }
00139 #endif
00140
00141
00142 #endif

```

10.37 SPI.h File Reference

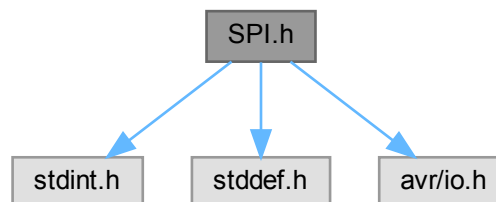
This file provides an interface to SPI subsystem available on the AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega) microcontrollers.

```

#include <stdint.h>
#include <stddef.h>
#include <avr/io.h>

```

Include dependency graph for SPI.h:



Classes

- class [SPI::SPISettings](#)
A class that binds settings for configuring SPI transmissions.

Namespaces

- namespace [SPI](#)
This namespace bundles an interface to the SPI hardware subsystem on the AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega) microcontrollers. It provides logical cohesion for functions implement the Master portion of the SPI protocol and prevents namespace collisions.

Enumerations

- enum `SPI::ByteOrder` { `SPI::kLsbFirst` , `SPI::kMsbFirst` }
An enumeration that defines the byte order for multibyte SPI transmissions.
- enum `SPI::SpiMode` { `SPI::kSpiMode0` , `SPI::kSpiMode1` , `SPI::kSpiMode2` , `SPI::kSpiMode3` }
An enumeration that defines the modes available for SPI transmissions.

Functions

- void `SPI::enable` ()
Enable the SPI subsystem for transmission.
- void `SPI::disable` ()
Disable the SPI subsystem, precluding further transmissions.
- void `SPI::configure` (`SPISettings` settings)
Set the configuration of SPI subsystem to match the needs of the system you are going to communicate with.
- `uint8_t SPI::transmit` (`uint8_t` data)
Transmit a single byte using the SPI subsystem.
- `uint16_t SPI::transmit16` (`uint16_t` data)
Transmit a word-sized integer (two bytes) using the SPI subsystem. The order in which the bytes are sent is determined by the bit order configuration that has been set.
- `uint32_t SPI::transmit32` (`uint32_t` data)
Transmit a long-word-sized integer (four bytes) using the SPI subsystem. The order in which the bytes are sent is determined by the bit order configuration that has been set.
- void `SPI::transmit` (`uint8_t` *buffer, `size_t` count)
Transmit an array of bytes using the SPI subsystem. The bytes are transmitted in array order.

10.37.1 Detailed Description

This file provides an interface to SPI subsystem available on the AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega) microcontrollers.

10.38 SPI.h

[Go to the documentation of this file.](#)

```
00001 /*
00002  SPI.h - an interface to the SPI subsystem of the
00003  AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00004  This is part of the AVRTools library.
00005  Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00006  Various portions of this code adapted from Arduino SPI code that
00007  is Copyright (c) 2010 by Cristian Maglie, Copyright (c) 2014 by Paul Stoffregen,
00008  Copyright (c) 2014 by Matthijs Kooijman, and Copyright (c) 2014 by Andrew J. Kroll
00009  and licensed under the terms of either the GNU General Public License version 2
00010  or the GNU Lesser General Public License version 2.1.
00011
00012  This program is free software: you can redistribute it and/or modify
00013  it under the terms of the GNU Lesser General Public License as published by
00014  the Free Software Foundation, either version 3 of the License, or
00015  (at your option) any later version.
00016
00017  This program is distributed in the hope that it will be useful,
00018  but WITHOUT ANY WARRANTY; without even the implied warranty of
```



```

00019     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00020     GNU Lesser General Public License for more details.
00021
00022     You should have received a copy of the GNU Lesser General Public License
00023     along with this program.  If not, see <http://www.gnu.org/licenses/>.
00024 */
00025
00026
00036 #ifndef SPI_h
00037 #define SPI_h
00038
00039 #include <stdint.h>
00040 #include <stddef.h>
00041
00042 #include <avr/io.h>
00043
00044
00109 namespace SPI
00110 {
00111
00115     enum ByteOrder
00116     {
00117         kLsbFirst    = 0,
00118         kMsbFirst    = 1
00119     };
00120
00121
00130     enum SpiMode
00131     {
00132         kSpiMode0    = 0x00,
00133         kSpiMode1    = 0x04,
00134         kSpiMode2    = 0x08,
00135         kSpiMode3    = 0x0C
00136     };
00137
00138
00158     class SPISettings
00159     {
00160     public:
00161
00179         SPISettings( uint32_t maxSpeed, uint8_t bitOrder, uint8_t dataMode )
00180         {
00181             if ( __builtin_constant_p( maxSpeed ) )
00182             {
00183                 initAlwaysInline( maxSpeed, bitOrder, dataMode );
00184             }
00185             else
00186             {
00187                 initMightInline( maxSpeed, bitOrder, dataMode );
00188             }
00189         }
00190
00191
00192
00199         SPISettings()
00200         {
00201             initAlwaysInline( 8000000, kMsbFirst, kSpiMode0 );
00202         }
00203
00204
00205
00212         uint8_t getSpcr() const
00213         {
00214             return mSpcr;
00215         }
00216
00217
00224         uint8_t getSpsr() const
00225         {
00226             return mSpsr;
00227         }
00228
00229
00230
00231
00232
00233     private:
00234
00235
00236         void initMightInline( uint32_t maxSpeed, uint8_t bitOrder, uint8_t dataMode )
00237         {

```

```

00238         initAlwaysInline( maxSpeed, bitOrder, dataMode );
00239     }
00240
00241
00242 #pragma GCC diagnostic push
00243 #pragma GCC diagnostic ignored "-Wunused-variable"
00244
00245     void initAlwaysInline( uint32_t maxSpeed, uint8_t bitOrder, uint8_t dataMode )
00246     __attribute__((__always_inline__))
00247     {
00248         /*
00249          * The following are internal constants
00250          */
00251         const uint8_t kSpiClockDiv4      = 0x00;
00252         const uint8_t kSpiClockDiv16     = 0x01;
00253         const uint8_t kSpiClockDiv64     = 0x02;
00254         const uint8_t kSpiClockDiv128    = 0x03;
00255         const uint8_t kSpiClockDiv2      = 0x04;
00256         const uint8_t kSpiClockDiv8      = 0x05;
00257         const uint8_t kSpiClockDiv32     = 0x06;
00258
00259         const uint8_t kSpiModeMask       = 0x0C;    // CPOL = bit 3, CPHA = bit 2 on SPCR
00260         const uint8_t kSpiClockMask      = 0x03;    // SPR1 = bit 1, SPR0 = bit 0 on SPCR
00261         const uint8_t kSpi2xClockMask    = 0x01;    // SPI2X = bit 0 on SPSR
00262
00263         // Clock settings are defined as follows. Note that this shows SPI2X
00264         // inverted, so the bits form increasing numbers. Also note that
00265         // fosc/64 appears twice
00266         // SPR1 SPR0 ~SPI2X Freq
00267         // 0 0 0 fosc/2
00268         // 0 0 1 fosc/4
00269         // 0 1 0 fosc/8
00270         // 0 1 1 fosc/16
00271         // 1 0 0 fosc/32
00272         // 1 0 1 fosc/64
00273         // 1 1 0 fosc/64
00274         // 1 1 1 fosc/128
00275
00276         // We find the fastest clock that is less than or equal to the
00277         // given clock rate. The clock divider that results in clock_setting
00278         // is 2 ^^ (clock_div + 1). If nothing is slow enough, we'll use the
00279         // slowest (128 == 2 ^^ 7, so clock_div = 6).
00280         uint8_t clockDiv;
00281
00282         // When the clock is known at compile time, use this if-then-else
00283         // cascade, which the compiler knows how to completely optimize
00284         // away. When clock is not known, use a loop instead, which generates
00285         // shorter code.
00286         if ( __builtin_constant_p( maxSpeed ) )
00287         {
00288             if ( maxSpeed >= F_CPU / 2 )
00289             {
00290                 clockDiv = 0;
00291             }
00292             else if ( maxSpeed >= F_CPU / 4 )
00293             {
00294                 clockDiv = 1;
00295             }
00296             else if ( maxSpeed >= F_CPU / 8 )
00297             {
00298                 clockDiv = 2;
00299             }
00300             else if ( maxSpeed >= F_CPU / 16 )
00301             {
00302                 clockDiv = 3;
00303             }
00304             else if ( maxSpeed >= F_CPU / 32 )
00305             {
00306                 clockDiv = 4;
00307             }
00308             else if ( maxSpeed >= F_CPU / 64 )
00309             {
00310                 clockDiv = 5;
00311             }
00312             else
00313             {
00314                 clockDiv = 6;
00315             }
00316         }
00317         else

```

```

00318         {
00319             uint32_t clockSetting = F_CPU / 2;
00320             clockDiv = 0;
00321             while ( clockDiv < 6 && maxSpeed < clockSetting )
00322             {
00323                 clockSetting /= 2;
00324                 clockDiv++;
00325             }
00326         }
00327
00328         // Compensate for the duplicate fosc/64
00329         if ( clockDiv == 6 )
00330         {
00331             clockDiv = 7;
00332         }
00333
00334         // Invert the SPI2X bit
00335         clockDiv ^= 0x1;
00336
00337         // Pack into the SPISettings class
00338         mSpcr = _BV(SPE)
00339             | _BV(MSTR)
00340             | ( (bitOrder == kLsbFirst) ? _BV(DORD) : 0 )
00341             | ( dataMode & kSpiModeMask )
00342             | ( (clockDiv >> 1) & kSpiClockMask );
00343
00344         mSpsr = clockDiv & kSpi2xClockMask;
00345     }
00346
00347 #pragma GCC diagnostic pop
00348
00349     uint8_t mSpcr;
00350     uint8_t mSpsr;
00351 };
00352
00353
00354
00355
00356
00371 void enable();
00372
00373
00374
00384 void disable();
00385
00386
00426 inline void configure( SPISettings settings )
00427 {
00428     SPCR = settings.getSpcr();
00429     SPSR = settings.getSpsr();
00430 }
00431
00432
00441 inline uint8_t transmit( uint8_t data )
00442 {
00443     SPDR = data;
00444     /*
00445      * The following NOP introduces a small delay that can prevent the wait
00446      * loop from iterating when running at the maximum speed. This gives
00447      * about 10% more speed, even if it seems counter-intuitive. At lower
00448      * speeds it is unnoticed.
00449      */
00450     asm volatile( "nop" );
00451     while ( !( SPSR & _BV(SPIF) ) )
00452         ; // wait
00453     return SPDR;
00454 }
00455
00456
00467 inline uint16_t transmit16( uint16_t data )
00468 {
00469     union
00470     {
00471         uint16_t val;
00472         struct
00473         {
00474             uint8_t lsb;
00475             uint8_t msb;
00476         };
00477     };
00478     in, out;

```

```

00479
00480     in.val = data;
00481
00482     if ( SPCR & _BV(DORD) )
00483     {
00484         SPDR = in.lsb;
00485         asm volatile( "nop" );           // See transmit( uint8_t ) function
00486         while ( !( SPSR & _BV(SPIF) ) )
00487             ;
00488         out.lsb = SPDR;
00489
00490         SPDR = in.msb;
00491         asm volatile( "nop" );
00492         while ( !( SPSR & _BV(SPIF) ) )
00493             ;
00494         out.msb = SPDR;
00495     }
00496     else
00497     {
00498         SPDR = in.msb;
00499         asm volatile( "nop" );           // See transmit( uint8_t ) function
00500         while ( !( SPSR & _BV(SPIF) ) )
00501             ;
00502         out.msb = SPDR;
00503         SPDR = in.lsb;
00504         asm volatile( "nop" );
00505         while ( !( SPSR & _BV(SPIF) ) )
00506             ;
00507         out.lsb = SPDR;
00508     }
00509
00510     return out.val;
00511 }
00512
00513
00524 uint32_t transmit32( uint32_t data );
00525
00526
00527
00539 inline void transmit( uint8_t* buffer, size_t count )
00540 {
00541     if ( count )
00542     {
00543         uint8_t* p = buffer;
00544         SPDR = *p;
00545
00546         while ( --count > 0 )
00547         {
00548             uint8_t out = *(p + 1);
00549             while ( !( SPSR & _BV(SPIF) ) )
00550                 ;
00551             uint8_t in = SPDR;
00552             SPDR = out;
00553             *p++ = in;
00554         }
00555
00556         while ( !(SPSR & _BV(SPIF) ) )
00557             ;
00558         *p = SPDR;
00559     }
00560 }
00561
00562
00563 } // End namespace
00564
00565 #endif
00566

```

10.39 SystemClock.h File Reference

Include this file to use the functions that instantiate and access a system clock that counts elapsed milliseconds.

Functions

- void `initSystemClock` ()
This function initializes a system clock that tracks elapsed milliseconds.
- void `delayMicroseconds` (unsigned int us)
Delay a certain number of microseconds.
- void `delayMilliseconds` (unsigned long ms)
Delay a certain number of milliseconds.
- void `delay` (unsigned long ms)
Delay a certain number of milliseconds.
- unsigned long `micros` ()
Return the number of elapsed microseconds since the system clock was turned on.
- unsigned long `millis` ()
Return the number of elapsed milliseconds since the system clock was turned on.

10.39.1 Detailed Description

Include this file to use the functions that instantiate and access a system clock that counts elapsed milliseconds.

To use these functions, include `SystemClock.h` in your source code and link against `SystemClock.cpp`.

Note

Linking against `SystemClock.cpp` installs a interrupt function on `timer0`. This interrupt routine is installed regardless of whether the system clock is actually initialized or not. If you have other uses for `timer0`, do not use `SystemClock` functions and do not link against `SystemClock.cpp`.

10.39.2 Function Documentation

10.39.2.1 `delay()`

```
void delay (
    unsigned long ms ) [inline]
```

Delay a certain number of milliseconds.

This inline function is a synonym for `delayMilliseconds()`; it is provided for compatibility with the standard Arduino library.

- `m` the number of milliseconds to delay.

10.39.2.2 delayMicroseconds()

```
void delayMicroseconds (
    unsigned int us )
```

Delay a certain number of microseconds.

- `us` the number of microseconds to delay.

Note

This function only works for CPU clocks running at either 8 MHz, 12 MHz, or 16 MHz.

10.39.2.3 delayMilliseconds()

```
void delayMilliseconds (
    unsigned long ms )
```

Delay a certain number of milliseconds.

- `m` the number of milliseconds to delay.

10.39.2.4 initSystemClock()

```
void initSystemClock ( )
```

This function initializes a system clock that tracks elapsed milliseconds.

The system clock uses `timer0`, so you cannot use `timer0` for other functions if you use the system clock functionality.

Note

Linking against `SystemClock.cpp` installs a interrupt function on `timer0`. This interrupt routine is installed regardless of whether the system clock is actually initialized or not. If you have other uses for `timer0`, do not use `SystemClock` functions and do not link against `SystemClock.cpp`.

10.39.2.5 micros()

```
unsigned long micros ( )
```

Return the number of elapsed microseconds since the system clock was turned on.

The microsecond count will overflow back to zero in approximately 70 minutes.

Returns

the number of elapsed microseconds.

10.39.2.6 millis()

```
unsigned long millis ( )
```

Return the number of elapsed milliseconds since the system clock was turned on.

Returns

the number of elapsed milliseconds.

10.40 SystemClock.h

[Go to the documentation of this file.](#)

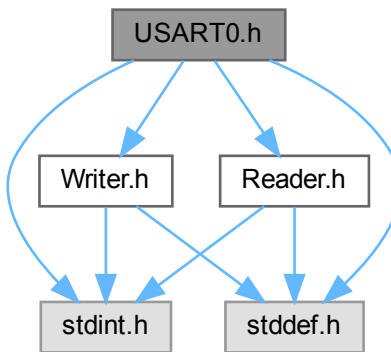
```
00001 /*
00002     SystemClock.h - Functions to initialize and use a system clock
00003     on AVR chips that is compatible with Arduino.
00004     For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00005     This is part of the AVRTools library.
00006     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00007     Functions delayMicroseconds() and delayMilliseconds() adapted from Arduino code that
00008     is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00009
00010     This program is free software: you can redistribute it and/or modify
00011     it under the terms of the GNU Lesser General Public License as published by
00012     the Free Software Foundation, either version 3 of the License, or
00013     (at your option) any later version.
00014
00015     This program is distributed in the hope that it will be useful,
00016     but WITHOUT ANY WARRANTY; without even the implied warranty of
00017     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00018     GNU Lesser General Public License for more details.
00019
00020     You should have received a copy of the GNU Lesser General Public License
00021     along with this program. If not, see <http://www.gnu.org/licenses/>.
00022 */
00023
00024
00025
00042 #ifndef SystemClock_h
00043 #define SystemClock_h
00044
00045
00046
00047
00048 #define clockCyclesPerMicrosecond() ( F_CPU / 1000000L )
00049 #define clockCyclesToMicroseconds( a ) ( (a) / clockCyclesPerMicrosecond() )
00050 #define microsecondsToClockCycles( a ) ( (a) * clockCyclesPerMicrosecond() )
00051
00052
00065 void initSystemClock();
00066
00067
00077 void delayMicroseconds( unsigned int us );
00078
00079
00087 void delayMilliseconds( unsigned long ms );
00088
00089
00100 inline void delay( unsigned long ms )
00101 { delayMilliseconds( ms ); }
00102
00103
00113 unsigned long micros();
00114
00115
00123 unsigned long millis();
00124
00125
00126
00127 #endif
```

10.41 USART0.h File Reference

This file provides functions that offer high-level interfaces to USART0 hardware, which is available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560).

```
#include "Writer.h"
#include "Reader.h"
#include <stdint.h>
#include <stddef.h>
```

Include dependency graph for USART0.h:



Classes

- class [Serial0](#)

Provides a high-end interface to serial communications using USART0.

Namespaces

- namespace [USART0](#)

This namespace bundles a high-level buffered interface to the USART0 hardware. It provides logical cohesion and prevents namespace collisions.

Enumerations

- enum [UsartSerialConfiguration](#) {
[kSerial_5N1](#) , [kSerial_6N1](#) , [kSerial_7N1](#) , [kSerial_8N1](#) ,
[kSerial_5N2](#) , [kSerial_6N2](#) , [kSerial_7N2](#) , [kSerial_8N2](#) ,
[kSerial_5E1](#) , [kSerial_6E1](#) , [kSerial_7E1](#) , [kSerial_8E1](#) ,
[kSerial_5E2](#) , [kSerial_6E2](#) , [kSerial_7E2](#) , [kSerial_8E2](#) ,
[kSerial_5O1](#) , [kSerial_6O1](#) , [kSerial_7O1](#) , [kSerial_8O1](#) ,
[kSerial_5O2](#) , [kSerial_6O2](#) , [kSerial_7O2](#) , [kSerial_8O2](#) }

This enum lists serial configuration in terms of data bits, parity, and stop bits.

Functions

- void `USART0::start` (unsigned long baudRate, `UsartSerialConfiguration` config=`kSerial_8N1`)
Initialize USART0 for buffered, asynchronous serial communications using interrupts.
- void `USART0::stop` ()
Stops buffered serial communications using interrupts on USART0.
- size_t `USART0::write` (char c)
Write a single byte to the transmit buffer.
- size_t `USART0::write` (const char *c)
Write a null-terminated string to the transmit buffer.
- size_t `USART0::write` (const char *c, size_t n)
Write a character array of given size to the transmit buffer.
- size_t `USART0::write` (const uint8_t *c, size_t n)
Write a byte array of given size to the transmit buffer.
- void `USART0::flush` ()
Flush transmit buffer.
- int `USART0::peek` ()
Examine the next character in the receive buffer without removing it from the buffer.
- int `USART0::read` ()
Return the next character in the receive buffer, removing it from the buffer.
- bool `USART0::available` ()
Determine if there is data in the receive buffer..

10.41.1 Detailed Description

This file provides functions that offer high-level interfaces to USART0 hardware, which is available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560).

These interfaces are buffered for both input and output and operate using interrupts associated with USART0. This means the transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART0 hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit and receive buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The receive buffer, however, will overwrite if it gets full. You must clear the receive buffer by reading it regularly when receiving significant amounts of data.

The sizes of the transmit and receive buffers can be set at compile time via macro constants. The default sizes are 32 bytes for the receive buffer and 64 bytes for the transmit buffer. To change these, define the macros `USART0_RX_BUFFER_SIZE` (for the receive buffer) and `USART0_TX_BUFFER_SIZE` (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file `USART0.cpp`.

Two interfaces are provided. `USART0` is a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontrollers. However, `USART0` is limited to transmitting and receiving byte and character streams.

`Serial0` is the most advanced and capable interface to the USART0 hardware. `Serial0` provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously.

To use these functions, include `USART0.h` in your source code and link against `USART0.cpp`.

Note

Linking against USART0.cpp installs interrupt functions for transmit and receive on USART0 (interrupts USART←_UDRE and USART_RX on Arduino Uno/ATmega328; interrupts USART0_UDRE and USART0_RX on Arduino Mega/ATmega2560). You cannot use the minimal interface to USART0 (from USARTMinimal.h) if you link against USART0.cpp. In particular, do *not* call initUSART0() or clearUSART0() if you link against USART0.cpp.

10.41.2 Enumeration Type Documentation**10.41.2.1 UsartSerialConfiguration**

```
enum UsartSerialConfiguration
```

This enum lists serial configuration in terms of data bits, parity, and stop bits.

The format is kSerial_XYZ where

- X = the number of data bits
- Y = N, E, or O; where N = none, E = even, and O = odd
- Z = the number of stop bits

Enumerator

kSerial_5N1	5 data bits, no parity, 1 stop bit
kSerial_6N1	6 data bits, no parity, 1 stop bit
kSerial_7N1	7 data bits, no parity, 1 stop bit
kSerial_8N1	8 data bits, no parity, 1 stop bit
kSerial_5N2	5 data bits, no parity, 2 stop bits
kSerial_6N2	6 data bits, no parity, 2 stop bits
kSerial_7N2	7 data bits, no parity, 2 stop bits
kSerial_8N2	8 data bits, no parity, 2 stop bits
kSerial_5E1	5 data bits, even parity, 1 stop bit
kSerial_6E1	6 data bits, even parity, 1 stop bit
kSerial_7E1	7 data bits, even parity, 1 stop bit
kSerial_8E1	8 data bits, even parity, 1 stop bit
kSerial_5E2	5 data bits, even parity, 2 stop bits
kSerial_6E2	6 data bits, even parity, 2 stop bits
kSerial_7E2	7 data bits, even parity, 2 stop bits
kSerial_8E2	8 data bits, even parity, 2 stop bits
kSerial_5O1	5 data bits, odd parity, 1 stop bit
kSerial_6O1	6 data bits, odd parity, 1 stop bit
kSerial_7O1	7 data bits, odd parity, 1 stop bit
kSerial_8O1	8 data bits, odd parity, 1 stop bit
kSerial_5O2	5 data bits, odd parity, 2 stop bits
kSerial_6O2	6 data bits, odd parity, 2 stop bits
kSerial_7O2	7 data bits, odd parity, 2 stop bits
kSerial_8O2	8 data bits, odd parity, 2 stop bits

10.42 USART0.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      USART0.h - Functions and classes to use USART0 on AVR systems for
00003      serial I/O (includes buffering).
00004      For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
00005      This is part of the AVRTools library.
00006      Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00007      Functions readlong() and readFloat() adapted from Arduino code that
00008      is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00009
00010      This program is free software: you can redistribute it and/or modify
00011      it under the terms of the GNU Lesser General Public License as published by
00012      the Free Software Foundation, either version 3 of the License, or
00013      (at your option) any later version.
00014
00015      This program is distributed in the hope that it will be useful,
00016      but WITHOUT ANY WARRANTY; without even the implied warranty of
00017      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00018      GNU Lesser General Public License for more details.
00019
00020      You should have received a copy of the GNU Lesser General Public License
00021      along with this program. If not, see <http://www.gnu.org/licenses/>.
00022  */
00023
00067  #ifndef USART0_h
00068  #define USART0_h
00069
00070  #include "Writer.h"
00071  #include "Reader.h"
00072
00073  #include <stdint.h>
00074  #include <stddef.h>
00075
00076
00077  #ifndef USART_SERIAL_CONFIG
00078  #define USART_SERIAL_CONFIG
00079
00090  enum UsartSerialConfiguration
00091  {
00092      kSerial_5N1 = 0x00,
00093      kSerial_6N1 = 0x02,
00094      kSerial_7N1 = 0x04,
00095      kSerial_8N1 = 0x06,
00096      kSerial_5N2 = 0x08,
00097      kSerial_6N2 = 0x0A,
00098      kSerial_7N2 = 0x0C,
00099      kSerial_8N2 = 0x0E,
00100      kSerial_5E1 = 0x20,
00101      kSerial_6E1 = 0x22,
00102      kSerial_7E1 = 0x24,
00103      kSerial_8E1 = 0x26,
00104      kSerial_5E2 = 0x28,
00105      kSerial_6E2 = 0x2A,
00106      kSerial_7E2 = 0x2C,
00107      kSerial_8E2 = 0x2E,
00108      kSerial_5O1 = 0x30,
00109      kSerial_6O1 = 0x32,
00110      kSerial_7O1 = 0x34,
00111      kSerial_8O1 = 0x36,
00112      kSerial_5O2 = 0x38,
00113      kSerial_6O2 = 0x3A,
00114      kSerial_7O2 = 0x3C,
00115      kSerial_8O2 = 0x3E
00116  };
00117
00118  #endif
00119
00120
00121
00122
00128  namespace USART0
00129  {
00130
00144      void start( unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1 );
00145
00146
00157      void stop();

```

```
00158
00159
00160
00175     size_t write( char c );
00176
00177
00192     size_t write( const char* c );
00193
00194
00211     size_t write( const char* c, size_t n );
00212
00213
00230     size_t write( const uint8_t* c, size_t n );
00231
00232
00241     void flush();
00242
00243
00251     int peek();
00252
00253
00261     int read();
00262
00263
00271     bool available();
00272 };
00273
00274
00275
00276
00277
00278
00279
00280
00295 class Serial0 : public Writer, public Reader
00296 {
00297 public:
00298
00311     void start( unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1 )
00312     { USART0::start( baudRate, config ); }
00313
00314
00324     void stop()
00325     { USART0::stop(); }
00326
00327
00328
00329
00330
00339     virtual size_t write( char c );
00340
00349     virtual size_t write( const char* str );
00350
00360     virtual size_t write( const char* buffer, size_t size );
00361
00371     virtual size_t write( const uint8_t* buffer, size_t size );
00372
00378     virtual void flush();
00379
00380
00381
00382     // Virtual functions from Reader
00383
00391     virtual int read();
00392
00393
00401     virtual int peek();
00402
00403
00410     virtual bool available();
00411 };
00412
00413
00414
00415 #endif
00416
```

10.43 USART0Minimal.h File Reference

This file provides functions that provide a minimalist interface to [USART0](#) available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560).

Functions

- void [initUSART0](#) (unsigned long baudRate)
Initialize USART0 for serial receive and transmit.
- void [transmitUSART0](#) (unsigned char data)
Transmit a single byte on USART0.
- void [transmitUSART0](#) (const char *data)
Transmit a null-terminated string on USART0.
- unsigned char [receiveUSART0](#) ()
Receive a byte on USART0.
- void [releaseUSART0](#) ()
Release USART0, making pins 0 and 1 again available for non-USART use.

10.43.1 Detailed Description

This file provides functions that provide a minimalist interface to [USART0](#) available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560).

These functions are minimalist in the following sense:

- They only send single bytes or zero-terminated character strings.
- They only receive single characters.
- They do not use the USART-related interrupts.
- They determine when the USART is ready to send by polling the relevant register bit.
- They determine when the USART has received data by polling the relevant register bit.

To use these functions, include [USART0Minimal.h](#) in your source code and link against USART0Minimal.cpp.

For a more advanced [USART0](#) interface, consider using either the [USART0](#) or [Serial0](#) interfaces. Both of these are available by including [USART0.h](#) instead of [USART0Minimal.h](#).

10.43.2 Function Documentation

10.43.2.1 `initUSART0()`

```
void initUSART0 (
    unsigned long baudRate )
```

Initialize USART0 for serial receive and transmit.

USART0 is tied to pins 0 (RX) and 1 (TX) on both Arduino Uno (ATmega328 pins PD0, PD1) and Arduino Mega (ATmega2560 pins PE0, PE1).

Communications are configured for 8 data bits, no parity, and 1 stop bit.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).

10.43.2.2 `receiveUSART0()`

```
unsigned char receiveUSART0 ( )
```

Receive a byte on USART0.

You must first initialize USART0 by calling [initUSART0\(\)](#).

This function blocks until the USART receives a byte.

Returns

the byte received.

10.43.2.3 `releaseUSART0()`

```
void releaseUSART0 ( )
```

Release USART0, making pins 0 and 1 again available for non-USART use.

After calling this function, you cannot read or write to the USART unless you first call [initUSART0\(\)](#).

10.43.2.4 `transmitUSART0()` [1/2]

```
void transmitUSART0 (
    const char * data )
```

Transmit a null-terminated string on USART0.

You must first initialize USART0 by calling [initUSART0\(\)](#).

This function blocks until the USART becomes available and all the bytes can be transmitted.

- `data` the null-terminated string to be transmitted.

10.43.2.5 transmitUSART0() [2/2]

```
void transmitUSART0 (
    unsigned char data )
```

Transmit a single byte on USART0.

You must first initialize USART0 by calling `initUSART0()`.

This function blocks until the USART becomes available and the byte can be transmitted.

- data the byte to be transmitted.

10.44 USART0Minimal.h

[Go to the documentation of this file.](#)

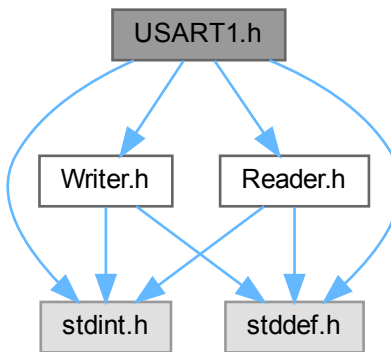
```
00001 /*
00002  USART0Minimal.h - Minimal, light-weight functions to use
00003  USART0 available on AVR ATmega328p (Arduino Uno) and
00004  ATmega2560 (Arduino Mega) processors (no buffering).
00005  This is part of the AVRTools library.
00006  Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00007  Functions readlong() and readFloat() adapted from Arduino code that
00008  is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00009
00010  This program is free software: you can redistribute it and/or modify
00011  it under the terms of the GNU Lesser General Public License as published by
00012  the Free Software Foundation, either version 3 of the License, or
00013  (at your option) any later version.
00014
00015  This program is distributed in the hope that it will be useful,
00016  but WITHOUT ANY WARRANTY; without even the implied warranty of
00017  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00018  GNU Lesser General Public License for more details.
00019
00020  You should have received a copy of the GNU Lesser General Public License
00021  along with this program. If not, see <http://www.gnu.org/licenses/>.
00022 */
00023
00024
00025
00048 #ifndef USART0Minimal_h
00049 #define USART0Minimal_h
00050
00051
00052
00066 void initUSART0( unsigned long baudRate );
00067
00068
00079 void transmitUSART0( unsigned char data );
00080
00081
00092 void transmitUSART0( const char* data );
00093
00094
00105 unsigned char receiveUSART0();
00106
00107
00115 void releaseUSART0();
00116
00117
00118
00119
00120 #endif
```

10.45 USART1.h File Reference

This file provides functions that offer high-level interfaces to USART1 hardware, which is available on Arduino Mega (ATMega2560).

```
#include "Writer.h"
#include "Reader.h"
#include <stdint.h>
#include <stddef.h>
```

Include dependency graph for USART1.h:



Classes

- class [Serial1](#)

Provides a high-end interface to serial communications using USART1.

Namespaces

- namespace [USART1](#)

This namespace bundles a high-level buffered interface to the USART1 hardware. It provides logical cohesion and prevents namespace collisions.

Enumerations

- enum [UsartSerialConfiguration](#) {
[kSerial_5N1](#) , [kSerial_6N1](#) , [kSerial_7N1](#) , [kSerial_8N1](#) ,
[kSerial_5N2](#) , [kSerial_6N2](#) , [kSerial_7N2](#) , [kSerial_8N2](#) ,
[kSerial_5E1](#) , [kSerial_6E1](#) , [kSerial_7E1](#) , [kSerial_8E1](#) ,
[kSerial_5E2](#) , [kSerial_6E2](#) , [kSerial_7E2](#) , [kSerial_8E2](#) ,
[kSerial_5O1](#) , [kSerial_6O1](#) , [kSerial_7O1](#) , [kSerial_8O1](#) ,
[kSerial_5O2](#) , [kSerial_6O2](#) , [kSerial_7O2](#) , [kSerial_8O2](#) }

This enum lists serial configuration in terms of data bits, parity, and stop bits.

Functions

- void [USART1::start](#) (unsigned long baudRate, [UsartSerialConfiguration](#) config=[kSerial_8N1](#))
Initialize USART1 for buffered, asynchronous serial communications using interrupts.
- void [USART1::stop](#) ()
Stops buffered serial communications using interrupts on USART1.
- size_t [USART1::write](#) (char c)
Write a single byte to the transmit buffer.
- size_t [USART1::write](#) (const char *c)
Write a null-terminated string to the transmit buffer.
- size_t [USART1::write](#) (const char *c, size_t n)
Write a character array of given size to the transmit buffer.
- size_t [USART1::write](#) (const uint8_t *c, size_t n)
Write a byte array of given size to the transmit buffer.
- void [USART1::flush](#) ()
Flush transmit buffer.
- int [USART1::peek](#) ()
Examine the next character in the receive buffer without removing it from the buffer.
- int [USART1::read](#) ()
Return the next character in the receive buffer, removing it from the buffer.
- bool [USART1::available](#) ()
Determine if there is data in the receive buffer..

10.45.1 Detailed Description

This file provides functions that offer high-level interfaces to USART1 hardware, which is available on Arduino Mega (ATMega2560).

These interfaces are buffered for both input and output and operate using interrupts associated with USART1. This means the transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART1 hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit and receive buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The receive buffer, however, will overwrite if it gets full. You must clear the receive buffer by reading it regularly when receiving significant amounts of data.

The sizes of the transmit and receive buffers can be set at compile time via macro constants. The default sizes are 32 bytes for the receive buffer and 64 bytes for the transmit buffer. To change these, define the macros `USART1_RX_BUFFER_SIZE` (for the receive buffer) and `USART1_TX_BUFFER_SIZE` (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file `USART1.cpp`.

Two interfaces are provided. [USART1](#) is a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontrollers. However, [USART1](#) is limited to transmitting and receiving byte and character streams.

[Serial1](#) is the most advanced and capable interface to the USART1 hardware. [Serial1](#) provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously.

To use these functions, include [USART1.h](#) in your source code and link against `USART1.cpp`.

Note

Linking against `USART1.cpp` installs interrupt functions for transmit and receive on USART1 (interrupts `USART1←_UDRE` and `USART1_RX` on Arduino Mega/ATmega2560). You cannot use the minimal interface to USART1 (from `USARTMinimal.h`) if you link against `USART1.cpp`. In particular, do *not* call `initUSART1()` or `clearUSART1()` if you link against `USART1.cpp`.

10.45.2 Enumeration Type Documentation

10.45.2.1 UsartSerialConfiguration

```
enum UsartSerialConfiguration
```

This enum lists serial configuration in terms of data bits, parity, and stop bits.

The format is `kSerial_XYZ` where

- X = the number of data bits
- Y = N, E, or O; where N = none, E = even, and O = odd
- Z = the number of stop bits

Enumerator

<code>kSerial_5N1</code>	5 data bits, no parity, 1 stop bit
<code>kSerial_6N1</code>	6 data bits, no parity, 1 stop bit
<code>kSerial_7N1</code>	7 data bits, no parity, 1 stop bit
<code>kSerial_8N1</code>	8 data bits, no parity, 1 stop bit
<code>kSerial_5N2</code>	5 data bits, no parity, 2 stop bits
<code>kSerial_6N2</code>	6 data bits, no parity, 2 stop bits
<code>kSerial_7N2</code>	7 data bits, no parity, 2 stop bits
<code>kSerial_8N2</code>	8 data bits, no parity, 2 stop bits
<code>kSerial_5E1</code>	5 data bits, even parity, 1 stop bit
<code>kSerial_6E1</code>	6 data bits, even parity, 1 stop bit
<code>kSerial_7E1</code>	7 data bits, even parity, 1 stop bit
<code>kSerial_8E1</code>	8 data bits, even parity, 1 stop bit
<code>kSerial_5E2</code>	5 data bits, even parity, 2 stop bits
<code>kSerial_6E2</code>	6 data bits, even parity, 2 stop bits
<code>kSerial_7E2</code>	7 data bits, even parity, 2 stop bits
<code>kSerial_8E2</code>	8 data bits, even parity, 2 stop bits
<code>kSerial_5O1</code>	5 data bits, odd parity, 1 stop bit
<code>kSerial_6O1</code>	6 data bits, odd parity, 1 stop bit
<code>kSerial_7O1</code>	7 data bits, odd parity, 1 stop bit
<code>kSerial_8O1</code>	8 data bits, odd parity, 1 stop bit
<code>kSerial_5O2</code>	5 data bits, odd parity, 2 stop bits
<code>kSerial_6O2</code>	6 data bits, odd parity, 2 stop bits
<code>kSerial_7O2</code>	7 data bits, odd parity, 2 stop bits
<code>kSerial_8O2</code>	8 data bits, odd parity, 2 stop bits

10.46 USART1.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      USART1.h - Functions and classes to use USART1 on AVR systems for
00003      serial I/O (includes buffering).
00004      For AVR ATmega2560 (Arduino Mega).
00005      This is part of the AVRTools library.
00006      Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00007      Functions readlong() and readFloat() adapted from Arduino code that
00008      is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00009
00010      This program is free software: you can redistribute it and/or modify
00011      it under the terms of the GNU Lesser General Public License as published by
00012      the Free Software Foundation, either version 3 of the License, or
00013      (at your option) any later version.
00014
00015      This program is distributed in the hope that it will be useful,
00016      but WITHOUT ANY WARRANTY; without even the implied warranty of
00017      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00018      GNU Lesser General Public License for more details.
00019
00020      You should have received a copy of the GNU Lesser General Public License
00021      along with this program. If not, see <http://www.gnu.org/licenses/>.
00022  */
00023
00024
00025  #if !defined(__AVR_ATmega2560__)
00026  #error "USART1 doesn't exist on ATmega328p (Arduino Uno); you can only use this on ATmega2560 (Arduino
00027  Mega)."
00028  #endif
00029
00030
00074  #ifndef USART1_h
00075  #define USART1_h
00076
00077  #include "Writer.h"
00078  #include "Reader.h"
00079
00080  #include <stdint.h>
00081  #include <stddef.h>
00082
00083
00084  #ifndef USART_SERIAL_CONFIG
00085  #define USART_SERIAL_CONFIG
00086
00097  enum UsartSerialConfiguration
00098  {
00099      kSerial_5N1 = 0x00,
00100      kSerial_6N1 = 0x02,
00101      kSerial_7N1 = 0x04,
00102      kSerial_8N1 = 0x06,
00103      kSerial_5N2 = 0x08,
00104      kSerial_6N2 = 0x0A,
00105      kSerial_7N2 = 0x0C,
00106      kSerial_8N2 = 0x0E,
00107      kSerial_5E1 = 0x20,
00108      kSerial_6E1 = 0x22,
00109      kSerial_7E1 = 0x24,
00110      kSerial_8E1 = 0x26,
00111      kSerial_5E2 = 0x28,
00112      kSerial_6E2 = 0x2A,
00113      kSerial_7E2 = 0x2C,
00114      kSerial_8E2 = 0x2E,
00115      kSerial_5O1 = 0x30,
00116      kSerial_6O1 = 0x32,
00117      kSerial_7O1 = 0x34,
00118      kSerial_8O1 = 0x36,
00119      kSerial_5O2 = 0x38,
00120      kSerial_6O2 = 0x3A,
00121      kSerial_7O2 = 0x3C,
00122      kSerial_8O2 = 0x3E
00123  };
00124
00125  #endif
00126
00127
00128

```

```
00129
00135 namespace USART1
00136 {
00137
00151     void start( unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1 );
00152
00153
00164     void stop();
00165
00166
00167
00182     size_t write( char c );
00183
00184
00199     size_t write( const char* c );
00200
00201
00218     size_t write( const char* c, size_t n );
00219
00220
00237     size_t write( const uint8_t* c, size_t n );
00238
00239
00248     void flush();
00249
00250
00258     int peek();
00259
00260
00268     int read();
00269
00270
00278     bool available();
00279 };
00280
00281
00282
00283
00284
00285
00286
00287
00302 class Serial1 : public Writer, public Reader
00303 {
00304 public:
00305
00318     void start( unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1 )
00319     { USART1::start( baudRate, config ); }
00320
00321
00331     void stop()
00332     { USART1::stop(); }
00333
00334
00335
00336
00337
00346     virtual size_t write( char c );
00347
00356     virtual size_t write( const char* str );
00357
00367     virtual size_t write( const char* buffer, size_t size );
00368
00378     virtual size_t write( const uint8_t* buffer, size_t size );
00379
00385     virtual void flush();
00386
00387
00388
00389     // Virtual functions from Reader
00390
00398     virtual int read();
00399
00400
00408     virtual int peek();
00409
00410
00417     virtual bool available();
00418 };
00419
00420
```

```
00421
00422 #endif
00423
```

10.47 USART1Minimal.h File Reference

This file provides functions that provide a minimalist interface to [USART1](#) available on the Arduino Mega (ATmega2560).

Functions

- void [initUSART1](#) (unsigned long baudRate)
Initialize [USART1](#) for serial receive and transmit.
- void [transmitUSART1](#) (unsigned char data)
Transmit a single byte on [USART1](#).
- void [transmitUSART1](#) (const char *data)
Transmit a null-terminated string on [USART1](#).
- unsigned char [receiveUSART1](#) ()
Receive a byte on [USART1](#).
- void [releaseUSART1](#) ()
Release [USART1](#), making pins 0 and 1 again available for non-USART use.

10.47.1 Detailed Description

This file provides functions that provide a minimalist interface to [USART1](#) available on the Arduino Mega (ATmega2560).

These functions are minimalist in the following sense:

- They only send single bytes or zero-terminated character strings.
- They only receive single characters.
- They do not use the USART-related interrupts.
- They determine when the USART is ready to send by polling the relevant register bit.
- They determine when the USART has received data by polling the relevant register bit.

To use these functions, include [USART1Minimal.h](#) in your source code and link against USART1Minimal.cpp.

For a more advanced [USART1](#) interface, consider using either the [USART1](#) or [Serial1](#) interfaces. Both of these are available by including [USART1.h](#) instead of [USART1Minimal.h](#).

10.47.2 Function Documentation

10.47.2.1 `initUSART1()`

```
void initUSART1 (
    unsigned long baudRate )
```

Initialize [USART1](#) for serial receive and transmit.

[USART1](#) is tied to pins 18 (TX) and 19 (RX) on Arduino Mega (ATmega2560 pins PD3, PD2).

Communications are configured for 8 data bits, no parity, and 1 stop bit.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).

Note

This function is only available on Arduino Mega (ATmega2560).

10.47.2.2 `receiveUSART1()`

```
unsigned char receiveUSART1 ( )
```

Receive a byte on [USART1](#).

You must first initialize [USART1](#) by calling `initUSART1()`.

This function blocks until the USART receives a byte.

Returns

the byte received.

Note

This function is only available on Arduino Mega (ATmega2560).

10.47.2.3 `releaseUSART1()`

```
void releaseUSART1 ( )
```

Release [USART1](#), making pins 0 and 1 again available for non-USART use.

After calling this function, you cannot read or write to the USART unless you first call `initUSART1()`.

Note

This function is only available on Arduino Mega (ATmega2560).

10.47.2.4 transmitUSART1() [1/2]

```
void transmitUSART1 (
    const char * data )
```

Transmit a null-terminated string on [USART1](#).

You must first initialize [USART1](#) by calling [initUSART1\(\)](#).

This function blocks until the USART becomes available and all the bytes can be transmitted.

- `data` the null-terminated string to be transmitted.

Note

This function is only available on Arduino Mega (ATmega2560).

10.47.2.5 transmitUSART1() [2/2]

```
void transmitUSART1 (
    unsigned char data )
```

Transmit a single byte on [USART1](#).

You must first initialize [USART1](#) by calling [initUSART1\(\)](#).

This function blocks until the USART becomes available and the byte can be transmitted.

- `data` the byte to be transmitted.

Note

This function is only available on Arduino Mega (ATmega2560).

10.48 USART1Minimal.h

[Go to the documentation of this file.](#)

```

00001 /*
00002     USART1Minimal.h - Minimal, light-weight functions to use
00003     USART1 available on ATmega2560 (Arduino Mega) processors (no buffering).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00006     Functions readlong() and readFloat() adapted from Arduino code that
00007     is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00008
00009     This program is free software: you can redistribute it and/or modify
00010     it under the terms of the GNU Lesser General Public License as published by
00011     the Free Software Foundation, either version 3 of the License, or
00012     (at your option) any later version.
00013
00014     This program is distributed in the hope that it will be useful,
00015     but WITHOUT ANY WARRANTY; without even the implied warranty of
00016     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00017     GNU Lesser General Public License for more details.
00018
00019     You should have received a copy of the GNU Lesser General Public License
00020     along with this program. If not, see <http://www.gnu.org/licenses/>.
00021 */
00022
00023
00024
00047 #ifndef USART1Minimal_h
00048 #define USART1Minimal_h
00049
00050
00051
00052 #if defined(__AVR_ATmega2560__)
00053
00054
00055
00070 void initUSART1( unsigned long baudRate );
00071
00072
00085 void transmitUSART1( unsigned char data );
00086
00087
00100 void transmitUSART1( const char* data );
00101
00102
00115 unsigned char receiveUSART1();
00116
00117
00127 void releaseUSART1();
00128
00129
00130
00131 #endif
00132
00133
00134
00135 #endif

```

10.49 USART2.h File Reference

This file provides functions that offer high-level interfaces to USART2 hardware, which is available on Arduino Mega (ATmega2560).

```

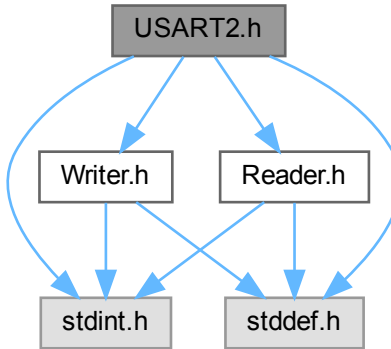
#include "Writer.h"
#include "Reader.h"
#include <stdint.h>

```



```
#include <stddef.h>
```

Include dependency graph for USART2.h:



Classes

- class [Serial2](#)

Provides a high-end interface to serial communications using USART2.

Namespaces

- namespace [USART2](#)

This namespace bundles a high-level buffered interface to the USART2 hardware. It provides logical cohesion and prevents namespace collisions.

Enumerations

- enum [UsartSerialConfiguration](#) {
[kSerial_5N1](#) , [kSerial_6N1](#) , [kSerial_7N1](#) , [kSerial_8N1](#) ,
[kSerial_5N2](#) , [kSerial_6N2](#) , [kSerial_7N2](#) , [kSerial_8N2](#) ,
[kSerial_5E1](#) , [kSerial_6E1](#) , [kSerial_7E1](#) , [kSerial_8E1](#) ,
[kSerial_5E2](#) , [kSerial_6E2](#) , [kSerial_7E2](#) , [kSerial_8E2](#) ,
[kSerial_5O1](#) , [kSerial_6O1](#) , [kSerial_7O1](#) , [kSerial_8O1](#) ,
[kSerial_5O2](#) , [kSerial_6O2](#) , [kSerial_7O2](#) , [kSerial_8O2](#) }

This enum lists serial configuration in terms of data bits, parity, and stop bits.

Functions

- void `USART2::start` (unsigned long baudRate, `UsartSerialConfiguration` config=`kSerial_8N1`)
Initialize USART2 for buffered, asynchronous serial communications using interrupts.
- void `USART2::stop` ()
Stops buffered serial communications using interrupts on USART2.
- size_t `USART2::write` (char c)
Write a single byte to the transmit buffer.
- size_t `USART2::write` (const char *c)
Write a null-terminated string to the transmit buffer.
- size_t `USART2::write` (const char *c, size_t n)
Write a character array of given size to the transmit buffer.
- size_t `USART2::write` (const uint8_t *c, size_t n)
Write a byte array of given size to the transmit buffer.
- void `USART2::flush` ()
Flush transmit buffer.
- int `USART2::peek` ()
Examine the next character in the receive buffer without removing it from the buffer.
- int `USART2::read` ()
Return the next character in the receive buffer, removing it from the buffer.
- bool `USART2::available` ()
Determine if there is data in the receive buffer..

10.49.1 Detailed Description

This file provides functions that offer high-level interfaces to USART2 hardware, which is available on Arduino Mega (ATMega2560).

These interfaces are buffered for both input and output and operate using interrupts associated with USART2. This means the transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART2 hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit and receive buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The receive buffer, however, will overwrite if it gets full. You must clear the receive buffer by reading it regularly when receiving significant amounts of data.

The sizes of the transmit and receive buffers can be set at compile time via macro constants. The default sizes are 32 bytes for the receive buffer and 64 bytes for the transmit buffer. To change these, define the macros `USART2_RX_BUFFER_SIZE` (for the receive buffer) and `USART2_TX_BUFFER_SIZE` (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file `USART2.cpp`.

Two interfaces are provided. `USART2` is a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontrollers. However, `USART2` is limited to transmitting and receiving byte and character streams.

`Serial2` is the most advanced and capable interface to the USART2 hardware. `Serial2` provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously.

To use these functions, include `USART2.h` in your source code and link against `USART2.cpp`.

Note

Linking against USART2.cpp installs interrupt functions for transmit and receive on USART2 (interrupts USART2←_UDRE and USART2_RX on Arduino Mega/ATmega2560). You cannot use the minimal interface to USART2 (from USARTMinimal.h) if you link against USART2.cpp. In particular, do *not* call initUSART2() or clearUSART2() if you link against USART2.cpp.

10.49.2 Enumeration Type Documentation**10.49.2.1 UsartSerialConfiguration**

```
enum UsartSerialConfiguration
```

This enum lists serial configuration in terms of data bits, parity, and stop bits.

The format is kSerial_XYZ where

- X = the number of data bits
- Y = N, E, or O; where N = none, E = even, and O = odd
- Z = the number of stop bits

Enumerator

kSerial_5N1	5 data bits, no parity, 1 stop bit
kSerial_6N1	6 data bits, no parity, 1 stop bit
kSerial_7N1	7 data bits, no parity, 1 stop bit
kSerial_8N1	8 data bits, no parity, 1 stop bit
kSerial_5N2	5 data bits, no parity, 2 stop bits
kSerial_6N2	6 data bits, no parity, 2 stop bits
kSerial_7N2	7 data bits, no parity, 2 stop bits
kSerial_8N2	8 data bits, no parity, 2 stop bits
kSerial_5E1	5 data bits, even parity, 1 stop bit
kSerial_6E1	6 data bits, even parity, 1 stop bit
kSerial_7E1	7 data bits, even parity, 1 stop bit
kSerial_8E1	8 data bits, even parity, 1 stop bit
kSerial_5E2	5 data bits, even parity, 2 stop bits
kSerial_6E2	6 data bits, even parity, 2 stop bits
kSerial_7E2	7 data bits, even parity, 2 stop bits
kSerial_8E2	8 data bits, even parity, 2 stop bits
kSerial_5O1	5 data bits, odd parity, 1 stop bit
kSerial_6O1	6 data bits, odd parity, 1 stop bit
kSerial_7O1	7 data bits, odd parity, 1 stop bit
kSerial_8O1	8 data bits, odd parity, 1 stop bit
kSerial_5O2	5 data bits, odd parity, 2 stop bits
kSerial_6O2	6 data bits, odd parity, 2 stop bits
kSerial_7O2	7 data bits, odd parity, 2 stop bits
kSerial_8O2	8 data bits, odd parity, 2 stop bits

10.50 USART2.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      USART2.h - Functions and classes to use USART2 on AVR systems for
00003      serial I/O (includes buffering).
00004      For AVR ATmega2560 (Arduino Mega).
00005      This is part of the AVRTools library.
00006      Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00007      Functions readlong() and readFloat() adapted from Arduino code that
00008      is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00009
00010      This program is free software: you can redistribute it and/or modify
00011      it under the terms of the GNU Lesser General Public License as published by
00012      the Free Software Foundation, either version 3 of the License, or
00013      (at your option) any later version.
00014
00015      This program is distributed in the hope that it will be useful,
00016      but WITHOUT ANY WARRANTY; without even the implied warranty of
00017      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00018      GNU Lesser General Public License for more details.
00019
00020      You should have received a copy of the GNU Lesser General Public License
00021      along with this program. If not, see <http://www.gnu.org/licenses/>.
00022  */
00023
00024
00025  #if !defined(__AVR_ATmega2560__)
00026  #error "USART2 doesn't exist on ATmega328p (Arduino Uno); you can only use this on ATmega2560 (Arduino
00027  Mega)."
00028  #endif
00029
00030
00074  #ifndef USART2_h
00075  #define USART2_h
00076
00077  #include "Writer.h"
00078  #include "Reader.h"
00079
00080  #include <stdint.h>
00081  #include <stddef.h>
00082
00083
00084  #ifndef USART_SERIAL_CONFIG
00085  #define USART_SERIAL_CONFIG
00086
00097  enum UsartSerialConfiguration
00098  {
00099      kSerial_5N1 = 0x00,
00100      kSerial_6N1 = 0x02,
00101      kSerial_7N1 = 0x04,
00102      kSerial_8N1 = 0x06,
00103      kSerial_5N2 = 0x08,
00104      kSerial_6N2 = 0x0A,
00105      kSerial_7N2 = 0x0C,
00106      kSerial_8N2 = 0x0E,
00107      kSerial_5E1 = 0x20,
00108      kSerial_6E1 = 0x22,
00109      kSerial_7E1 = 0x24,
00110      kSerial_8E1 = 0x26,
00111      kSerial_5E2 = 0x28,
00112      kSerial_6E2 = 0x2A,
00113      kSerial_7E2 = 0x2C,
00114      kSerial_8E2 = 0x2E,
00115      kSerial_5O1 = 0x30,
00116      kSerial_6O1 = 0x32,
00117      kSerial_7O1 = 0x34,
00118      kSerial_8O1 = 0x36,
00119      kSerial_5O2 = 0x38,
00120      kSerial_6O2 = 0x3A,
00121      kSerial_7O2 = 0x3C,
00122      kSerial_8O2 = 0x3E
00123  };
00124
00125  #endif
00126
00127
00128

```

```
00129
00135 namespace USART2
00136 {
00137
00151     void start( unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1 );
00152
00153
00164     void stop();
00165
00166
00167
00182     size_t write( char c );
00183
00184
00199     size_t write( const char* c );
00200
00201
00218     size_t write( const char* c, size_t n );
00219
00220
00237     size_t write( const uint8_t* c, size_t n );
00238
00239
00248     void flush();
00249
00250
00258     int peek();
00259
00260
00268     int read();
00269
00270
00278     bool available();
00279 };
00280
00281
00282
00283
00284
00285
00286
00287
00302 class Serial2 : public Writer, public Reader
00303 {
00304 public:
00305
00318     void start( unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1 )
00319     { USART2::start( baudRate, config ); }
00320
00321
00331     void stop()
00332     { USART2::stop(); }
00333
00334
00335
00336
00337
00346     virtual size_t write( char c );
00347
00356     virtual size_t write( const char* str );
00357
00367     virtual size_t write( const char* buffer, size_t size );
00368
00378     virtual size_t write( const uint8_t* buffer, size_t size );
00379
00385     virtual void flush();
00386
00387
00388
00389     // Virtual functions from Reader
00390
00398     virtual int read();
00399
00400
00408     virtual int peek();
00409
00410
00417     virtual bool available();
00418 };
00419
00420
```

```
00421
00422 #endif
00423
```

10.51 USART2Minimal.h File Reference

This file provides functions that provide a minimalist interface to [USART2](#) available on the Arduino Mega (ATmega2560).

Functions

- void [initUSART2](#) (unsigned long baudRate)
Initialize [USART2](#) for serial receive and transmit.
- void [transmitUSART2](#) (unsigned char data)
Transmit a single byte on [USART2](#).
- void [transmitUSART2](#) (const char *data)
Transmit a null-terminated string on [USART2](#).
- unsigned char [receiveUSART2](#) ()
Receive a byte on [USART2](#).
- void [releaseUSART2](#) ()
Release [USART2](#), making pins 0 and 1 again available for non-USART use.

10.51.1 Detailed Description

This file provides functions that provide a minimalist interface to [USART2](#) available on the Arduino Mega (ATmega2560).

These functions are minimalist in the following sense:

- They only send single bytes or zero-terminated character strings.
- They only receive single characters.
- They do not use the USART-related interrupts.
- They determine when the USART is ready to send by polling the relevant register bit.
- They determine when the USART has received data by polling the relevant register bit.

To use these functions, include [USART2Minimal.h](#) in your source code and link against USART2Minimal.cpp.

For a more advanced [USART2](#) interface, consider using either the [USART2](#) or [Serial2](#) interfaces. Both of these are available by including [USART2.h](#) instead of [USART2Minimal.h](#).

10.51.2 Function Documentation

10.51.2.1 initUSART2()

```
void initUSART2 (
    unsigned long baudRate )
```

Initialize [USART2](#) for serial receive and transmit.

[USART2](#) is tied to pins 16 (TX) and 17 (RX) on Arduino Mega (ATmega2560 pins PH1, PH0).

Communications are configured for 8 data bits, no parity, and 1 stop bit.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).

Note

This function is only available on Arduino Mega (ATmega2560).

10.51.2.2 receiveUSART2()

```
unsigned char receiveUSART2 ( )
```

Receive a byte on [USART2](#).

You must first initialize [USART2](#) by calling `initUSART2()`.

This function blocks until the USART receives a byte.

Returns

the byte received.

Note

This function is only available on Arduino Mega (ATmega2560).

10.51.2.3 releaseUSART2()

```
void releaseUSART2 ( )
```

Release [USART2](#), making pins 0 and 1 again available for non-USART use.

After calling this function, you cannot read or write to the USART unless you first call `initUSART2()`.

Note

This function is only available on Arduino Mega (ATmega2560).

10.51.2.4 `transmitUSART2()` [1/2]

```
void transmitUSART2 (
    const char * data )
```

Transmit a null-terminated string on [USART2](#).

You must first initialize [USART2](#) by calling [initUSART2\(\)](#).

This function blocks until the USART becomes available and all the bytes can be transmitted.

- `data` the null-terminated string to be transmitted.

Note

This function is only available on Arduino Mega (ATmega2560).

10.51.2.5 `transmitUSART2()` [2/2]

```
void transmitUSART2 (
    unsigned char data )
```

Transmit a single byte on [USART2](#).

You must first initialize [USART2](#) by calling [initUSART2\(\)](#).

This function blocks until the USART becomes available and the byte can be transmitted.

- `data` the byte to be transmitted.

Note

This function is only available on Arduino Mega (ATmega2560).

10.52 USART2Minimal.h

[Go to the documentation of this file.](#)

```
00001 /*
00002     USART2Minimal.h - Minimal, light-weight functions to use
00003     USART2 available on ATmega2560 (Arduino Mega) processors (no buffering).
00004     This is part of the AVRTools library.
00005     Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006     Functions readlong() and readFloat() adapted from Arduino code that
00007     is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00008
00009     This program is free software: you can redistribute it and/or modify
00010     it under the terms of the GNU Lesser General Public License as published by
00011     the Free Software Foundation, either version 3 of the License, or
00012     (at your option) any later version.
00013
00014     This program is distributed in the hope that it will be useful,
00015     but WITHOUT ANY WARRANTY; without even the implied warranty of
00016     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00017     GNU Lesser General Public License for more details.
00018
00019     You should have received a copy of the GNU Lesser General Public License
00020     along with this program. If not, see <http://www.gnu.org/licenses/>.
00021 */
00022
00023
00024
00047 #ifndef USART2Minimal_h
00048 #define USART2Minimal_h
00049
00050
00051
00052 #if defined(__AVR_ATmega2560__)
00053
00054
00055
00070 void initUSART2( unsigned long baudRate );
00071
00072
00085 void transmitUSART2( unsigned char data );
00086
00087
00100 void transmitUSART2( const char* data );
00101
00102
00115 unsigned char receiveUSART2();
00116
00117
00127 void releaseUSART2();
00128
00129
00130
00131 #endif
00132
00133
00134
00135 #endif
```

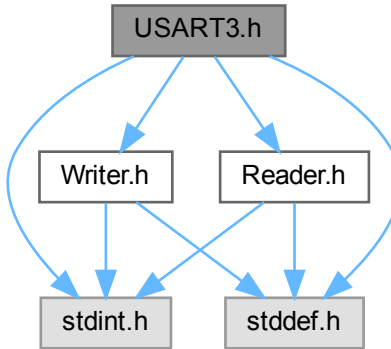
10.53 USART3.h File Reference

This file provides functions that offer high-level interfaces to USART3 hardware, which is available on Arduino Mega (ATmega2560).

```
#include "Writer.h"
#include "Reader.h"
#include <stdint.h>
```

```
#include <stddef.h>
```

Include dependency graph for USART3.h:



Classes

- class [Serial3](#)

Provides a high-end interface to serial communications using USART3.

Namespaces

- namespace [USART3](#)

This namespace bundles a high-level buffered interface to the USART3 hardware. It provides logical cohesion and prevents namespace collisions.

Enumerations

- enum [UsartSerialConfiguration](#) {
[kSerial_5N1](#) , [kSerial_6N1](#) , [kSerial_7N1](#) , [kSerial_8N1](#) ,
[kSerial_5N2](#) , [kSerial_6N2](#) , [kSerial_7N2](#) , [kSerial_8N2](#) ,
[kSerial_5E1](#) , [kSerial_6E1](#) , [kSerial_7E1](#) , [kSerial_8E1](#) ,
[kSerial_5E2](#) , [kSerial_6E2](#) , [kSerial_7E2](#) , [kSerial_8E2](#) ,
[kSerial_5O1](#) , [kSerial_6O1](#) , [kSerial_7O1](#) , [kSerial_8O1](#) ,
[kSerial_5O2](#) , [kSerial_6O2](#) , [kSerial_7O2](#) , [kSerial_8O2](#) }

This enum lists serial configuration in terms of data bits, parity, and stop bits.

Functions

- void `USART3::start` (unsigned long baudRate, `UsartSerialConfiguration` config=`kSerial_8N1`)
Initialize USART3 for buffered, asynchronous serial communications using interrupts.
- void `USART3::stop` ()
Stops buffered serial communications using interrupts on USART3.
- size_t `USART3::write` (char c)
Write a single byte to the transmit buffer.
- size_t `USART3::write` (const char *c)
Write a null-terminated string to the transmit buffer.
- size_t `USART3::write` (const char *c, size_t n)
Write a character array of given size to the transmit buffer.
- size_t `USART3::write` (const uint8_t *c, size_t n)
Write a byte array of given size to the transmit buffer.
- void `USART3::flush` ()
Flush transmit buffer.
- int `USART3::peek` ()
Examine the next character in the receive buffer without removing it from the buffer.
- int `USART3::read` ()
Return the next character in the receive buffer, removing it from the buffer.
- bool `USART3::available` ()
Determine if there is data in the receive buffer..

10.53.1 Detailed Description

This file provides functions that offer high-level interfaces to USART3 hardware, which is available on Arduino Mega (ATMega2560).

These interfaces are buffered for both input and output and operate using interrupts associated with USART3. This means the transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART3 hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit and receive buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The receive buffer, however, will overwrite if it gets full. You must clear the receive buffer by reading it regularly when receiving significant amounts of data.

The sizes of the transmit and receive buffers can be set at compile time via macro constants. The default sizes are 32 bytes for the receive buffer and 64 bytes for the transmit buffer. To change these, define the macros `USART3_RX_BUFFER_SIZE` (for the receive buffer) and `USART3_TX_BUFFER_SIZE` (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file `USART3.cpp`.

Two interfaces are provided. `USART3` is a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontrollers. However, `USART3` is limited to transmitting and receiving byte and character streams.

`Serial3` is the most advanced and capable interface to the USART3 hardware. `Serial3` provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously.

To use these functions, include `USART3.h` in your source code and link against `USART3.cpp`.

Note

Linking against USART3.cpp installs interrupt functions for transmit and receive on USART3 (interrupts USART3←_UDRE and USART3_RX on Arduino Mega/ATmega2560). You cannot use the minimal interface to USART3 (from USARTMinimal.h) if you link against USART3.cpp. In particular, do *not* call initUSART3() or clearUSART3() if you link against USART3.cpp.

10.53.2 Enumeration Type Documentation**10.53.2.1 UsartSerialConfiguration**

```
enum UsartSerialConfiguration
```

This enum lists serial configuration in terms of data bits, parity, and stop bits.

The format is kSerial_XYZ where

- X = the number of data bits
- Y = N, E, or O; where N = none, E = even, and O = odd
- Z = the number of stop bits

Enumerator

kSerial_5N1	5 data bits, no parity, 1 stop bit
kSerial_6N1	6 data bits, no parity, 1 stop bit
kSerial_7N1	7 data bits, no parity, 1 stop bit
kSerial_8N1	8 data bits, no parity, 1 stop bit
kSerial_5N2	5 data bits, no parity, 2 stop bits
kSerial_6N2	6 data bits, no parity, 2 stop bits
kSerial_7N2	7 data bits, no parity, 2 stop bits
kSerial_8N2	8 data bits, no parity, 2 stop bits
kSerial_5E1	5 data bits, even parity, 1 stop bit
kSerial_6E1	6 data bits, even parity, 1 stop bit
kSerial_7E1	7 data bits, even parity, 1 stop bit
kSerial_8E1	8 data bits, even parity, 1 stop bit
kSerial_5E2	5 data bits, even parity, 2 stop bits
kSerial_6E2	6 data bits, even parity, 2 stop bits
kSerial_7E2	7 data bits, even parity, 2 stop bits
kSerial_8E2	8 data bits, even parity, 2 stop bits
kSerial_5O1	5 data bits, odd parity, 1 stop bit
kSerial_6O1	6 data bits, odd parity, 1 stop bit
kSerial_7O1	7 data bits, odd parity, 1 stop bit
kSerial_8O1	8 data bits, odd parity, 1 stop bit
kSerial_5O2	5 data bits, odd parity, 2 stop bits
kSerial_6O2	6 data bits, odd parity, 2 stop bits
kSerial_7O2	7 data bits, odd parity, 2 stop bits
kSerial_8O2	8 data bits, odd parity, 2 stop bits

10.54 USART3.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      USART3.h - Functions and classes to use USART3 on AVR systems for
00003      serial I/O (includes buffering).
00004      For AVR ATmega2560 (Arduino Mega).
00005      This is part of the AVRTools library.
00006      Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00007      Functions readlong() and readFloat() adapted from Arduino code that
00008      is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00009
00010      This program is free software: you can redistribute it and/or modify
00011      it under the terms of the GNU Lesser General Public License as published by
00012      the Free Software Foundation, either version 3 of the License, or
00013      (at your option) any later version.
00014
00015      This program is distributed in the hope that it will be useful,
00016      but WITHOUT ANY WARRANTY; without even the implied warranty of
00017      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00018      GNU Lesser General Public License for more details.
00019
00020      You should have received a copy of the GNU Lesser General Public License
00021      along with this program. If not, see <http://www.gnu.org/licenses/>.
00022  */
00023
00024
00025  #if !defined(__AVR_ATmega2560__)
00026  #error "USART3 doesn't exist on ATmega328p (Arduino Uno); you can only use this on ATmega2560 (Arduino
00027  Mega)."
00028  #endif
00029
00030
00074  #ifndef USART3_h
00075  #define USART3_h
00076
00077  #include "Writer.h"
00078  #include "Reader.h"
00079
00080  #include <stdint.h>
00081  #include <stddef.h>
00082
00083
00084  #ifndef USART_SERIAL_CONFIG
00085  #define USART_SERIAL_CONFIG
00086
00097  enum UsartSerialConfiguration
00098  {
00099      kSerial_5N1 = 0x00,
00100      kSerial_6N1 = 0x02,
00101      kSerial_7N1 = 0x04,
00102      kSerial_8N1 = 0x06,
00103      kSerial_5N2 = 0x08,
00104      kSerial_6N2 = 0x0A,
00105      kSerial_7N2 = 0x0C,
00106      kSerial_8N2 = 0x0E,
00107      kSerial_5E1 = 0x20,
00108      kSerial_6E1 = 0x22,
00109      kSerial_7E1 = 0x24,
00110      kSerial_8E1 = 0x26,
00111      kSerial_5E2 = 0x28,
00112      kSerial_6E2 = 0x2A,
00113      kSerial_7E2 = 0x2C,
00114      kSerial_8E2 = 0x2E,
00115      kSerial_5O1 = 0x30,
00116      kSerial_6O1 = 0x32,
00117      kSerial_7O1 = 0x34,
00118      kSerial_8O1 = 0x36,
00119      kSerial_5O2 = 0x38,
00120      kSerial_6O2 = 0x3A,
00121      kSerial_7O2 = 0x3C,
00122      kSerial_8O2 = 0x3E
00123  };
00124
00125  #endif
00126
00127
00128

```

```
00129
00135 namespace USART3
00136 {
00137
00151     void start( unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1 );
00152
00153
00164     void stop();
00165
00166
00167
00182     size_t write( char c );
00183
00184
00199     size_t write( const char* c );
00200
00201
00218     size_t write( const char* c, size_t n );
00219
00220
00237     size_t write( const uint8_t* c, size_t n );
00238
00239
00248     void flush();
00249
00250
00258     int peek();
00259
00260
00268     int read();
00269
00270
00278     bool available();
00279 };
00280
00281
00282
00283
00284
00285
00286
00287
00302 class Serial3 : public Writer, public Reader
00303 {
00304 public:
00305
00318     void start( unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1 )
00319     { USART3::start( baudRate, config ); }
00320
00321
00331     void stop()
00332     { USART3::stop(); }
00333
00334
00335
00336
00337
00346     virtual size_t write( char c );
00347
00356     virtual size_t write( const char* str );
00357
00367     virtual size_t write( const char* buffer, size_t size );
00368
00378     virtual size_t write( const uint8_t* buffer, size_t size );
00379
00385     virtual void flush();
00386
00387
00388
00389     // Virtual functions from Reader
00390
00398     virtual int read();
00399
00400
00408     virtual int peek();
00409
00410
00417     virtual bool available();
00418 };
00419
00420
```

```
00421
00422 #endif
00423
```

10.55 USART3Minimal.h File Reference

This file provides functions that provide a minimalist interface to [USART3](#) available on the Arduino Mega (ATmega2560).

Functions

- void [initUSART3](#) (unsigned long baudRate)
Initialize [USART3](#) for serial receive and transmit.
- void [transmitUSART3](#) (unsigned char data)
Transmit a single byte on [USART3](#).
- void [transmitUSART3](#) (const char *data)
Transmit a null-terminated string on [USART3](#).
- unsigned char [receiveUSART3](#) ()
Receive a byte on [USART3](#).
- void [releaseUSART3](#) ()
Release [USART3](#), making pins 0 and 1 again available for non-USART use.

10.55.1 Detailed Description

This file provides functions that provide a minimalist interface to [USART3](#) available on the Arduino Mega (ATmega2560).

These functions are minimalist in the following sense:

- They only send single bytes or zero-terminated character strings.
- They only receive single characters.
- They do not use the USART-related interrupts.
- They determine when the USART is ready to send by polling the relevant register bit.
- They determine when the USART has received data by polling the relevant register bit.

To use these functions, include [USART3Minimal.h](#) in your source code and link against USART3Minimal.cpp.

For a more advanced [USART3](#) interface, consider using either the [USART3](#) or [Serial3](#) interfaces. Both of these are available by including [USART3.h](#) instead of [USART3Minimal.h](#).

10.55.2 Function Documentation

10.55.2.1 initUSART3()

```
void initUSART3 (
    unsigned long baudRate )
```

Initialize [USART3](#) for serial receive and transmit.

[USART3](#) is tied to pins 14 (TX) and 15 (RX) on Arduino Mega (ATmega2560 pins PJ1, PJ0).

Communications are configured for 8 data bits, no parity, and 1 stop bit.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).

Note

This function is only available on Arduino Mega (ATmega2560).

10.55.2.2 receiveUSART3()

```
unsigned char receiveUSART3 ( )
```

Receive a byte on [USART3](#).

You must first initialize [USART3](#) by calling `initUSART3()`.

This function blocks until the USART receives a byte.

Returns

the byte received.

Note

This function is only available on Arduino Mega (ATmega2560).

10.55.2.3 releaseUSART3()

```
void releaseUSART3 ( )
```

Release [USART3](#), making pins 0 and 1 again available for non-USART use.

After calling this function, you cannot read or write to the USART unless you first call `initUSART3()`.

Note

This function is only available on Arduino Mega (ATmega2560).

10.55.2.4 transmitUSART3() [1/2]

```
void transmitUSART3 (
    const char * data )
```

Transmit a null-terminated string on [USART3](#).

You must first initialize [USART3](#) by calling [initUSART3\(\)](#).

This function blocks until the USART becomes available and all the bytes can be transmitted.

- `data` the null-terminated string to be transmitted.

Note

This function is only available on Arduino Mega (ATmega2560).

10.55.2.5 transmitUSART3() [2/2]

```
void transmitUSART3 (
    unsigned char data )
```

Transmit a single byte on [USART3](#).

You must first initialize [USART3](#) by calling [initUSART3\(\)](#).

This function blocks until the USART becomes available and the byte can be transmitted.

- `data` the byte to be transmitted.

Note

This function is only available on Arduino Mega (ATmega2560).

10.56 USART3Minimal.h

[Go to the documentation of this file.](#)

```

00001  /*
00002      USART3Minimal.h - Minimal, light-weight functions to use
00003      USART3 available on ATmega2560 (Arduino Mega) processors (no buffering).
00004      This is part of the AVRTools library.
00005      Copyright (c) 2015 Igor Mikolic-Torreira. All right reserved.
00006      Functions readlong() and readFloat() adapted from Arduino code that
00007      is Copyright (c) 2005-2006 David A. Mellis and licensed under LGPL.
00008
00009      This program is free software: you can redistribute it and/or modify
00010      it under the terms of the GNU Lesser General Public License as published by
00011      the Free Software Foundation, either version 3 of the License, or
00012      (at your option) any later version.
00013
00014      This program is distributed in the hope that it will be useful,
00015      but WITHOUT ANY WARRANTY; without even the implied warranty of
00016      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00017      GNU Lesser General Public License for more details.
00018
00019      You should have received a copy of the GNU Lesser General Public License
00020      along with this program. If not, see <http://www.gnu.org/licenses/>.
00021  */
00022
00023
00024
00047  #ifndef USART3Minimal_h
00048  #define USART3Minimal_h
00049
00050
00051
00052  #if defined(__AVR_ATmega2560__)
00053
00054
00055
00070  void initUSART3( unsigned long baudRate );
00071
00072
00085  void transmitUSART3( unsigned char data );
00086
00087
00100  void transmitUSART3( const char* data );
00101
00102
00115  unsigned char receiveUSART3();
00116
00117
00127  void releaseUSART3();
00128
00129  #endif
00130
00131
00132
00133  #endif

```

10.57 Writer.h File Reference

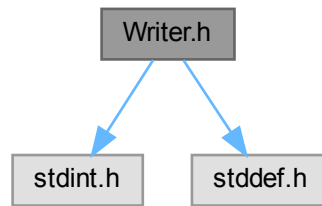
This file provides a generic interface to outgoing data streams of any kind. It is designed around how serial streams are generally used, but can be used with any system that requires converting strings and/or numbers into a sequential output of bytes.

```

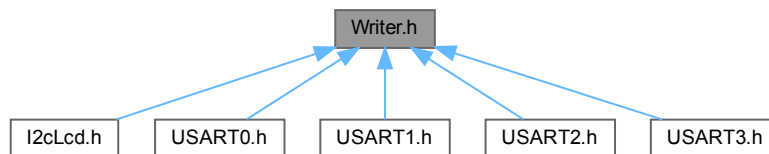
#include <stdint.h>
#include <stddef.h>

```

Include dependency graph for Writer.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [Writer](#)

This is an abstract class defining a generic interface to write numbers and strings to a sequential stream of bytes (such as a serial output device).

10.57.1 Detailed Description

This file provides a generic interface to outgoing data streams of any kind. It is designed around how serial streams are generally used, but can be used with any system that requires converting strings and/or numbers into a sequential output of bytes.

10.58 Writer.h

[Go to the documentation of this file.](#)

```

00001 /*
00002  * Writer.h - a base class for writing data.
00003  * For AVR ATmega328p (Arduino Uno) and ATmega2560 (Arduino Mega).
  
```

```

00004      This is part of the AVRTools library.
00005      Copyright (c) 2014 Igor Mikolic-Torreira. All right reserved.
00006      Functions printNumber() and printFloat() adapted from Arduino code that
00007      is Copyright (c) 2008 David A. Mellis and licensed under LGPL.
00008
00009      This program is free software: you can redistribute it and/or modify
00010      it under the terms of the GNU Lesser General Public License as published by
00011      the Free Software Foundation, either version 3 of the License, or
00012      (at your option) any later version.
00013
00014      This program is distributed in the hope that it will be useful,
00015      but WITHOUT ANY WARRANTY; without even the implied warranty of
00016      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00017      GNU Lesser General Public License for more details.
00018
00019      You should have received a copy of the GNU Lesser General Public License
00020      along with this program. If not, see <http://www.gnu.org/licenses/>.
00021  */
00022
00023
00034  #ifndef Writer_h
00035  #define Writer_h
00036
00037  #include <stdint.h>
00038  #include <stddef.h>
00039
00040
00041
00042  #ifndef SERIAL_OUTPUT_EOL
00043  #define SERIAL_OUTPUT_EOL    '\n'
00044  #endif
00045
00046
00047
00048
00063  class Writer
00064  {
00065  public:
00066
00067
00073      enum IntegerOutputBase
00074      {
00075          kBin    = 2,
00076          kOct    = 8,
00077          kDec    = 10,
00078          kHex    = 16
00079      };
00080
00081
00082
00090      virtual size_t write( char c ) = 0;
00091
00092
00100      virtual size_t write( const char* str ) = 0;
00101
00102
00111      virtual size_t write( const char* buffer, size_t size ) = 0;
00112
00113
00122      virtual size_t write( const uint8_t* buffer, size_t size ) = 0;
00123
00124
00129      virtual void flush() = 0;
00130
00131
00132
00133
00145      size_t print( const char* str, bool addLn = false );
00146
00147
00160      size_t print( const uint8_t* buf, size_t size, bool addLn = false );
00161
00162
00174      size_t print( char c, bool addLn = false );
00175
00176
00190      size_t print( int8_t n, int base = kDec, bool addLn = false );
00191
00192
00206      size_t print( uint8_t n, int base = kDec, bool addLn = false )
00207      { return print( static_cast<unsigned long>( n ), base, addLn ); }

```

```
00208
00222     size_t print( int n, int base = kDec, bool addLn = false );
00223
00224
00238     size_t print( unsigned int n, int base = kDec, bool addLn = false )
00239     { return print( static_cast<unsigned long>( n ), base, addLn ); }
00240
00241
00255     size_t print( long n, int base = kDec, bool addLn = false );
00256
00257
00271     size_t print( unsigned long n, int base = kDec, bool addLn = false );
00272
00273
00286     size_t print( double d, int digits = 2, bool addLn = false );
00287
00288
00297     size_t println( const char* str )                { return print( str, true ); }
00298
00308     size_t println( const uint8_t* buf, size_t size ) { return print( buf, size, true ); }
00309
00318     size_t println( char c )                        { return print( c, true ); }
00319
00332     size_t println( int8_t n, int base = kDec )      { return print( n, base, true ); }
00333
00346     size_t println( uint8_t n, int base = kDec )     { return print( n, base, true ); }
00347
00360     size_t println( int n, int base = kDec )        { return print( n, base, true ); }
00361
00374     size_t println( unsigned int n, int base = kDec ) { return print( n, base, true ); }
00375
00388     size_t println( long n, int base = kDec )       { return print( n, base, true ); }
00389
00402     size_t println( unsigned long n, int base = kDec ) { return print( n, base, true ); }
00403
00415     size_t println( double d, int digits = 2 )     { return print( d, digits, true ); }
00416
00420     size_t println();
00421
00422 private:
00423
00424     size_t printNumber( unsigned long n, uint8_t base );
00425     size_t printFloat( double d, uint8_t digits );
00426 };
00427
00428 #endif
00429
```


Index

- A2DVoltageReference
 - Analog2Digital.h, [198](#)
- abi.h, [195](#), [196](#)
- Advanced Features, [9](#)
- Analog2Digital.h, [196](#), [201](#)
 - A2DVoltageReference, [198](#)
 - initA2D, [199](#)
 - kA2dReference11V, [198](#)
 - kA2dReference256V, [198](#)
 - kA2dReferenceAREF, [198](#)
 - kA2dReferenceAVCC, [198](#)
 - readA2D, [199](#)
 - readGpioPinAnalog, [198](#)
 - readGpioPinAnalogV, [199](#)
 - setA2DVoltageReference, [200](#)
 - setA2DVoltageReference11V, [200](#)
 - setA2DVoltageReference256V, [200](#)
 - setA2DVoltageReferenceAREF, [201](#)
 - setA2DVoltageReferenceAVCC, [201](#)
- ArduinoMegaPins.h, [203](#), [204](#)
- ArduinoPins.h, [206](#), [207](#)
- ArduinoUnoPins.h, [207](#), [208](#)
- available
 - Reader, [93](#)
 - Serial0, [109](#)
 - Serial1, [128](#)
 - Serial2, [147](#)
 - Serial3, [166](#)
 - USART0, [57](#)
 - USART1, [61](#)
 - USART2, [65](#)
 - USART3, [69](#)
- AVRTools: A Library for the AVR ATmega328 and ATmega2560 Microcontrollers, [1](#)
- busy
 - I2cMaster, [37](#)
 - I2cSlave, [46](#)
- ByteOrder
 - SPI, [53](#)
- clearTimer0
 - Pwm.h, [249](#)
- clearTimer1
 - Pwm.h, [249](#)
- clearTimer2
 - Pwm.h, [250](#)
- clearTimer3
 - Pwm.h, [250](#)
- clearTimer4
 - Pwm.h, [250](#)
- clearTimer5
 - Pwm.h, [250](#)
- command
 - I2cLcd, [80](#)
- configure
 - SPI, [54](#)
- delay
 - SystemClock.h, [273](#)
- delayMicroseconds
 - SystemClock.h, [273](#)
- delayMilliseconds
 - SystemClock.h, [274](#)
- delayQuartersOfMicroSeconds
 - SimpleDelays.h, [265](#)
- delayTenthsOfSeconds
 - SimpleDelays.h, [265](#)
- delayWholeMilliseconds
 - SimpleDelays.h, [265](#)
- disable
 - SPI, [54](#)
- discardFromFront
 - RingBufferT < T, N, SIZE >, [103](#)
- displayBottomRow
 - I2cLcd, [80](#)
- displayTopRow
 - I2cLcd, [80](#)
- enable
 - SPI, [54](#)
- ExternalInterrupts
 - Interrupts, [48](#)
- ExternalOff
 - Interrupts::ExternalOff, [74](#)
- FAQ, [19](#)
- find
 - Reader, [93](#), [94](#)
 - Serial0, [110](#)
 - Serial1, [129](#)
 - Serial2, [148](#)

- Serial3, 167
- findUntil
 - Reader, 94
 - Serial0, 110
 - Serial1, 129
 - Serial2, 148
 - Serial3, 167
- flush
 - USART0, 57
 - USART1, 61
 - USART2, 65
 - USART3, 69
- freeMemoryBetweenHeapAndStack
 - MemUtils, 50
- freeSRAM
 - MemUtils, 50
- getFreeListStats
 - MemUtils, 50
- getGpioADC
 - GpioPinMacros.h, 212
- getGpioCOM
 - GpioPinMacros.h, 212
- getGpioDDR
 - GpioPinMacros.h, 213
- getGpioMASK
 - GpioPinMacros.h, 213
- getGpioOCR
 - GpioPinMacros.h, 213
- getGpioPIN
 - GpioPinMacros.h, 214
- getGpioPORT
 - GpioPinMacros.h, 214
- getGpioTCCR
 - GpioPinMacros.h, 214
- getSpcr
 - SPI::SPISettings, 183
- getSpsr
 - SPI::SPISettings, 183
- GpioPin
 - GpioPinMacros.h, 215
- GpioPinAnalog
 - GpioPinMacros.h, 215
- GpioPinMacros.h, 209, 222
 - getGpioADC, 212
 - getGpioCOM, 212
 - getGpioDDR, 213
 - getGpioMASK, 213
 - getGpioOCR, 213
 - getGpioPIN, 214
 - getGpioPORT, 214
 - getGpioTCCR, 214
 - GpioPin, 215
 - GpioPinAnalog, 215
 - GpioPinPwm, 215
 - isGpioPinModeInput, 216
 - isGpioPinModeInputV, 219
 - isGpioPinModeOutput, 216
 - isGpioPinModeOutputV, 219
 - kDigitalHigh, 219
 - kDigitalLow, 219
 - makeGpioVarFromGpioPin, 216
 - makeGpioVarFromGpioPinAnalog, 216
 - makeGpioVarFromGpioPinPwm, 217
 - readGpioPinDigital, 217
 - readGpioPinDigitalV, 220
 - setGpioPinHigh, 217
 - setGpioPinHighV, 220
 - setGpioPinLow, 218
 - setGpioPinLowV, 220
 - setGpioPinModeInput, 218
 - setGpioPinModeInputPullup, 218
 - setGpioPinModeInputPullupV, 220
 - setGpioPinModeInputV, 221
 - setGpioPinModeOutput, 218
 - setGpioPinModeOutputV, 221
 - writeGpioPinDigital, 219
 - writeGpioPinDigitalV, 221
- GpioPinPwm
 - GpioPinMacros.h, 215
- GpioPinVariable, 74
- I2cBusSpeed
 - I2cMaster, 35
 - I2cSlave, 45
- I2cLcd, 76
 - command, 80
 - displayBottomRow, 80
 - displayTopRow, 80
 - init, 81
 - IntegerOutputBase, 80
 - kBacklight_Blue, 80
 - kBacklight_Green, 80
 - kBacklight_Red, 79
 - kBacklight_Teal, 80
 - kBacklight_Violet, 80
 - kBacklight_White, 80
 - kBacklight_Yellow, 80
 - kBin, 80
 - kButton_Down, 79
 - kButton_Left, 79
 - kButton_Right, 79
 - kButton_Select, 79
 - kButton_Up, 79
 - kDec, 80
 - kHex, 80
 - kOct, 80
 - print, 81–84

- println, 85–88
- readButtons, 88
- setBacklight, 88
- setCursor, 89
- write, 89, 90
- I2cLcd.h, 226, 227
- I2cMaster, 33
 - busy, 37
 - I2cBusSpeed, 35
 - I2cPullups, 35
 - I2cSendErrorCodes, 36
 - I2cStatusCodes, 36
 - kl2cBusFast, 35
 - kl2cBusSlow, 35
 - kl2cCompletedOk, 36
 - kl2cErrMsgTooLong, 36
 - kl2cErrNullStatusPtr, 36
 - kl2cError, 36
 - kl2cErrReadWithoutStorage, 36
 - kl2cErrTxBufferFull, 36
 - kl2cErrWriteWithoutData, 36
 - kl2cInProgress, 36
 - kl2cNoError, 36
 - kl2cNotStarted, 36
 - kPullupsOff, 36
 - kPullupsOn, 36
 - pullups, 37
 - readAsync, 37
 - readSync, 38
 - start, 39
 - stop, 39
 - writeAsync, 39–41
 - writeSync, 42, 43
- I2cMaster.h, 230, 233
- I2cPullups
 - I2cMaster, 35
 - I2cSlave, 45
- I2cSendErrorCodes
 - I2cMaster, 36
- I2cSlave, 44
 - busy, 46
 - I2cBusSpeed, 45
 - I2cPullups, 45
 - I2cStatusCodes, 45
 - kl2cBusFast, 45
 - kl2cBusSlow, 45
 - kl2cCompletedOk, 46
 - kl2cError, 46
 - kl2cInProgress, 46
 - kl2cRxOverflow, 46
 - kl2cTxPartial, 46
 - kPullupsOff, 45
 - kPullupsOn, 45
 - processI2cMessage, 46
 - pullups, 47
 - start, 47
 - stop, 47
- I2cSlave.h, 235, 237
- I2cStatusCodes
 - I2cMaster, 36
 - I2cSlave, 45
- init
 - I2cLcd, 81
- initA2D
 - Analog2Digital.h, 199
- initPwmTimer0
 - Pwm.h, 251
- initPwmTimer1
 - Pwm.h, 251
- initPwmTimer2
 - Pwm.h, 252
- initPwmTimer3
 - Pwm.h, 252
- initPwmTimer4
 - Pwm.h, 252
- initPwmTimer5
 - Pwm.h, 253
- initSystem
 - InitSystem.h, 239
- InitSystem.h, 238, 239
 - initSystem, 239
- initSystemClock
 - SystemClock.h, 274
- initUSART0
 - USART0Minimal.h, 282
- initUSART1
 - USART1Minimal.h, 290
- initUSART2
 - USART2Minimal.h, 299
- initUSART3
 - USART3Minimal.h, 308
- IntegerOutputBase
 - I2cLcd, 80
 - Serial0, 109
 - Serial1, 128
 - Serial2, 147
 - Serial3, 166
 - Writer, 185
- Interrupts, 48
 - ExternalInterrupts, 48
 - kExternalInterrupt0, 49
 - kExternalInterrupt1, 49
 - kExternalInterrupt2, 49
 - kExternalInterrupt3, 49
 - kExternalInterrupt4, 49
 - kExternalInterrupt5, 49
 - kExternalInterrupt6, 49
 - kExternalInterrupt7, 49

- kExternalInterruptAll, [49](#)
- kPinChangeInterrupt0, [49](#)
- kPinChangeInterrupt1, [49](#)
- kPinChangeInterrupt2, [49](#)
- kPinChangeInterruptAll, [49](#)
- PinChangeInterrupts, [49](#)
- Interrupts::AllOff, [73](#)
- Interrupts::ExternalOff, [73](#)
 - ExternalOff, [74](#)
- Interrupts::PinChangeOff, [91](#)
 - PinChangeOff, [91](#)
- InterruptUtils.h, [239](#), [241](#)
- isEmpty
 - RingBuffer, [100](#)
 - RingBufferT< T, N, SIZE >, [103](#)
- isFull
 - RingBuffer, [100](#)
 - RingBufferT< T, N, SIZE >, [103](#)
- isGpioPinModeInput
 - GpioPinMacros.h, [216](#)
- isGpioPinModeInputV
 - GpioPinMacros.h, [219](#)
- isGpioPinModeOutput
 - GpioPinMacros.h, [216](#)
- isGpioPinModeOutputV
 - GpioPinMacros.h, [219](#)
- isNotEmpty
 - RingBuffer, [100](#)
 - RingBufferT< T, N, SIZE >, [104](#)
- isNotFull
 - RingBuffer, [101](#)
 - RingBufferT< T, N, SIZE >, [104](#)
- kA2dReference11V
 - Analog2Digital.h, [198](#)
- kA2dReference256V
 - Analog2Digital.h, [198](#)
- kA2dReferenceAREF
 - Analog2Digital.h, [198](#)
- kA2dReferenceAVCC
 - Analog2Digital.h, [198](#)
- kBacklight_Blue
 - I2cLcd, [80](#)
- kBacklight_Green
 - I2cLcd, [80](#)
- kBacklight_Red
 - I2cLcd, [79](#)
- kBacklight_Teal
 - I2cLcd, [80](#)
- kBacklight_Violet
 - I2cLcd, [80](#)
- kBacklight_White
 - I2cLcd, [80](#)
- kBacklight_Yellow
 - I2cLcd, [80](#)
- kBin
 - I2cLcd, [80](#)
 - Serial0, [109](#)
 - Serial1, [128](#)
 - Serial2, [147](#)
 - Serial3, [166](#)
 - Writer, [185](#)
- kButton_Down
 - I2cLcd, [79](#)
- kButton_Left
 - I2cLcd, [79](#)
- kButton_Right
 - I2cLcd, [79](#)
- kButton_Select
 - I2cLcd, [79](#)
- kButton_Up
 - I2cLcd, [79](#)
- kDec
 - I2cLcd, [80](#)
 - Serial0, [109](#)
 - Serial1, [128](#)
 - Serial2, [147](#)
 - Serial3, [166](#)
 - Writer, [185](#)
- kDigitalHigh
 - GpioPinMacros.h, [219](#)
- kDigitalLow
 - GpioPinMacros.h, [219](#)
- kExternalInterrupt0
 - Interrupts, [49](#)
- kExternalInterrupt1
 - Interrupts, [49](#)
- kExternalInterrupt2
 - Interrupts, [49](#)
- kExternalInterrupt3
 - Interrupts, [49](#)
- kExternalInterrupt4
 - Interrupts, [49](#)
- kExternalInterrupt5
 - Interrupts, [49](#)
- kExternalInterrupt6
 - Interrupts, [49](#)
- kExternalInterrupt7
 - Interrupts, [49](#)
- kExternalInterruptAll
 - Interrupts, [49](#)
- kHex
 - I2cLcd, [80](#)
 - Serial0, [109](#)
 - Serial1, [128](#)
 - Serial2, [147](#)
 - Serial3, [166](#)
 - Writer, [185](#)

- kl2cBusFast
 - I2cMaster, [35](#)
 - I2cSlave, [45](#)
- kl2cBusSlow
 - I2cMaster, [35](#)
 - I2cSlave, [45](#)
- kl2cCompletedOk
 - I2cMaster, [36](#)
 - I2cSlave, [46](#)
- kl2cErrMsgTooLong
 - I2cMaster, [36](#)
- kl2cErrNullStatusPtr
 - I2cMaster, [36](#)
- kl2cError
 - I2cMaster, [36](#)
 - I2cSlave, [46](#)
- kl2cErrReadWithoutStorage
 - I2cMaster, [36](#)
- kl2cErrTxBufferFull
 - I2cMaster, [36](#)
- kl2cErrWriteWithoutData
 - I2cMaster, [36](#)
- kl2cInProgress
 - I2cMaster, [36](#)
 - I2cSlave, [46](#)
- kl2cNoError
 - I2cMaster, [36](#)
- kl2cNotStarted
 - I2cMaster, [36](#)
- kl2cRxOverflow
 - I2cSlave, [46](#)
- kl2cTxPartial
 - I2cSlave, [46](#)
- kLsbFirst
 - SPI, [53](#)
- kMsbFirst
 - SPI, [53](#)
- kOct
 - I2cLcd, [80](#)
 - Serial0, [109](#)
 - Serial1, [128](#)
 - Serial2, [147](#)
 - Serial3, [166](#)
 - Writer, [185](#)
- kPinChangeInterrupt0
 - Interrupts, [49](#)
- kPinChangeInterrupt1
 - Interrupts, [49](#)
- kPinChangeInterrupt2
 - Interrupts, [49](#)
- kPinChangeInterruptAll
 - Interrupts, [49](#)
- kPullupsOff
 - I2cMaster, [36](#)
 - I2cSlave, [45](#)
- kPullupsOn
 - I2cMaster, [36](#)
 - I2cSlave, [45](#)
- kSerial_5E1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_5E2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_5N1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_5N2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_5O1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_5O2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_6E1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_6E2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_6N1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_6N2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)

- kSerial_6O1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_6O2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_7E1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_7E2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_7N1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_7N2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_7O1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_7O2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_8E1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_8E2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_8N1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
- USART3.h, [304](#)
- kSerial_8N2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_8O1
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSerial_8O2
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- kSpiMode0
 - SPI, [53](#)
- kSpiMode1
 - SPI, [53](#)
- kSpiMode2
 - SPI, [53](#)
- kSpiMode3
 - SPI, [53](#)
- makeGpioVarFromGpioPin
 - GpioPinMacros.h, [216](#)
- makeGpioVarFromGpioPinAnalog
 - GpioPinMacros.h, [216](#)
- makeGpioVarFromGpioPinPwm
 - GpioPinMacros.h, [217](#)
- memoryAvailableOnFreeList
 - MemUtils, [50](#)
- MemUtils, [49](#)
 - freeMemoryBetweenHeapAndStack, [50](#)
 - freeSRAM, [50](#)
 - getFreeListStats, [50](#)
 - memoryAvailableOnFreeList, [50](#)
 - resetHeap, [51](#)
- MemUtils.h, [243](#), [244](#)
- micros
 - SystemClock.h, [274](#)
- millis
 - SystemClock.h, [274](#)
- new.h, [245](#), [246](#)
- peek
 - Reader, [95](#)
 - RingBuffer, [101](#)
 - RingBufferT< T, N, SIZE >, [104](#)
 - Serial0, [111](#)
 - Serial1, [130](#)
 - Serial2, [149](#)
 - Serial3, [168](#)

- USART0, [57](#)
- USART1, [61](#)
- USART2, [65](#)
- USART3, [69](#)
- PinChangeInterrupts
 - Interrupts, [49](#)
- PinChangeOff
 - Interrupts::PinChangeOff, [91](#)
- print
 - I2cLcd, [81–84](#)
 - Serial0, [111–115](#)
 - Serial1, [130–134](#)
 - Serial2, [149–153](#)
 - Serial3, [168–172](#)
 - Writer, [186–189](#)
- println
 - I2cLcd, [85–88](#)
 - Serial0, [115–118](#)
 - Serial1, [134–137](#)
 - Serial2, [153–156](#)
 - Serial3, [172–175](#)
 - Writer, [190–193](#)
- processI2cMessage
 - I2cSlave, [46](#)
- pull
 - RingBuffer, [101](#)
 - RingBufferT< T, N, SIZE >, [104](#)
- pullups
 - I2cMaster, [37](#)
 - I2cSlave, [47](#)
- push
 - RingBuffer, [101](#)
 - RingBufferT< T, N, SIZE >, [105](#)
- Pwm.h, [246, 254](#)
 - clearTimer0, [249](#)
 - clearTimer1, [249](#)
 - clearTimer2, [250](#)
 - clearTimer3, [250](#)
 - clearTimer4, [250](#)
 - clearTimer5, [250](#)
 - initPwmTimer0, [251](#)
 - initPwmTimer1, [251](#)
 - initPwmTimer2, [252](#)
 - initPwmTimer3, [252](#)
 - initPwmTimer4, [252](#)
 - initPwmTimer5, [253](#)
 - writeGpioPinPwm, [249](#)
 - writeGpioPinPwmV, [253](#)
- read
 - Reader, [95](#)
 - Serial0, [118](#)
 - Serial1, [137](#)
 - Serial2, [156](#)
 - Serial3, [175](#)
 - USART0, [58](#)
 - USART1, [61](#)
 - USART2, [65](#)
 - USART3, [69](#)
- readA2D
 - Analog2Digital.h, [199](#)
- readAsync
 - I2cMaster, [37](#)
- readButtons
 - I2cLcd, [88](#)
- readBytes
 - Reader, [95, 96](#)
 - Serial0, [119](#)
 - Serial1, [138](#)
 - Serial2, [157](#)
 - Serial3, [176](#)
- readBytesUntil
 - Reader, [96](#)
 - Serial0, [119, 120](#)
 - Serial1, [138, 139](#)
 - Serial2, [157, 158](#)
 - Serial3, [176, 177](#)
- Reader, [92](#)
 - available, [93](#)
 - find, [93, 94](#)
 - findUntil, [94](#)
 - peek, [95](#)
 - read, [95](#)
 - readBytes, [95, 96](#)
 - readBytesUntil, [96](#)
 - readFloat, [97](#)
 - readLine, [97](#)
 - readLong, [98](#)
 - setTimeout, [98](#)
- Reader.h, [256, 257](#)
- readFloat
 - Reader, [97](#)
 - Serial0, [120, 121](#)
 - Serial1, [139, 140](#)
 - Serial2, [158, 159](#)
 - Serial3, [177, 178](#)
- readGpioPinAnalog
 - Analog2Digital.h, [198](#)
- readGpioPinAnalogV
 - Analog2Digital.h, [199](#)
- readGpioPinDigital
 - GpioPinMacros.h, [217](#)
- readGpioPinDigitalV
 - GpioPinMacros.h, [220](#)
- readLine
 - Reader, [97](#)
 - Serial0, [121](#)
 - Serial1, [140](#)

- Serial2, [159](#)
- Serial3, [178](#)
- readLong
 - Reader, [98](#)
 - Serial0, [121](#), [122](#)
 - Serial1, [140](#), [141](#)
 - Serial2, [159](#), [160](#)
 - Serial3, [178](#), [179](#)
- readSync
 - I2cMaster, [38](#)
- receiveUSART0
 - USART0Minimal.h, [282](#)
- receiveUSART1
 - USART1Minimal.h, [290](#)
- receiveUSART2
 - USART2Minimal.h, [299](#)
- receiveUSART3
 - USART3Minimal.h, [308](#)
- releaseUSART0
 - USART0Minimal.h, [282](#)
- releaseUSART1
 - USART1Minimal.h, [290](#)
- releaseUSART2
 - USART2Minimal.h, [299](#)
- releaseUSART3
 - USART3Minimal.h, [308](#)
- resetHeap
 - MemUtils, [51](#)
- RingBuffer, [99](#)
 - isEmpty, [100](#)
 - isFull, [100](#)
 - isNotEmpty, [100](#)
 - isNotFull, [101](#)
 - peek, [101](#)
 - pull, [101](#)
 - push, [101](#)
 - RingBuffer, [100](#)
- RingBuffer.h, [259](#), [260](#)
- RingBufferT< T, N, SIZE >, [102](#)
 - discardFromFront, [103](#)
 - isEmpty, [103](#)
 - isFull, [103](#)
 - isNotEmpty, [104](#)
 - isNotFull, [104](#)
 - peek, [104](#)
 - pull, [104](#)
 - push, [105](#)
- RingBufferT.h, [261](#)
- Serial0, [106](#)
 - available, [109](#)
 - find, [110](#)
 - findUntil, [110](#)
 - IntegerOutputBase, [109](#)
- kBin, [109](#)
- kDec, [109](#)
- kHex, [109](#)
- kOct, [109](#)
- peek, [111](#)
- print, [111–115](#)
- println, [115–118](#)
- read, [118](#)
- readBytes, [119](#)
- readBytesUntil, [119](#), [120](#)
- readFloat, [120](#), [121](#)
- readLine, [121](#)
- readLong, [121](#), [122](#)
- setTimeout, [122](#)
- start, [122](#)
- stop, [123](#)
- write, [123](#), [124](#)
- Serial1, [125](#)
 - available, [128](#)
 - find, [129](#)
 - findUntil, [129](#)
 - IntegerOutputBase, [128](#)
 - kBin, [128](#)
 - kDec, [128](#)
 - kHex, [128](#)
 - kOct, [128](#)
 - peek, [130](#)
 - print, [130–134](#)
 - println, [134–137](#)
 - read, [137](#)
 - readBytes, [138](#)
 - readBytesUntil, [138](#), [139](#)
 - readFloat, [139](#), [140](#)
 - readLine, [140](#)
 - readLong, [140](#), [141](#)
 - setTimeout, [141](#)
 - start, [141](#)
 - stop, [142](#)
 - write, [142](#), [143](#)
- Serial2, [144](#)
 - available, [147](#)
 - find, [148](#)
 - findUntil, [148](#)
 - IntegerOutputBase, [147](#)
 - kBin, [147](#)
 - kDec, [147](#)
 - kHex, [147](#)
 - kOct, [147](#)
 - peek, [149](#)
 - print, [149–153](#)
 - println, [153–156](#)
 - read, [156](#)
 - readBytes, [157](#)
 - readBytesUntil, [157](#), [158](#)

- readFloat, [158](#), [159](#)
- readLine, [159](#)
- readLong, [159](#), [160](#)
- setTimeout, [160](#)
- start, [160](#)
- stop, [161](#)
- write, [161](#), [162](#)
- Serial3, [163](#)
 - available, [166](#)
 - find, [167](#)
 - findUntil, [167](#)
 - IntegerOutputBase, [166](#)
 - kBin, [166](#)
 - kDec, [166](#)
 - kHex, [166](#)
 - kOct, [166](#)
 - peek, [168](#)
 - print, [168–172](#)
 - println, [172–175](#)
 - read, [175](#)
 - readBytes, [176](#)
 - readBytesUntil, [176](#), [177](#)
 - readFloat, [177](#), [178](#)
 - readLine, [178](#)
 - readLong, [178](#), [179](#)
 - setTimeout, [179](#)
 - start, [179](#)
 - stop, [180](#)
 - write, [180](#), [181](#)
- setA2DVoltageReference
 - Analog2Digital.h, [200](#)
- setA2DVoltageReference11V
 - Analog2Digital.h, [200](#)
- setA2DVoltageReference256V
 - Analog2Digital.h, [200](#)
- setA2DVoltageReferenceAREF
 - Analog2Digital.h, [201](#)
- setA2DVoltageReferenceAVCC
 - Analog2Digital.h, [201](#)
- setBacklight
 - I2cLcd, [88](#)
- setCursor
 - I2cLcd, [89](#)
- setGpioPinHigh
 - GpioPinMacros.h, [217](#)
- setGpioPinHighV
 - GpioPinMacros.h, [220](#)
- setGpioPinLow
 - GpioPinMacros.h, [218](#)
- setGpioPinLowV
 - GpioPinMacros.h, [220](#)
- setGpioPinModeInput
 - GpioPinMacros.h, [218](#)
- setGpioPinModeInputPullup
 - GpioPinMacros.h, [218](#)
- setGpioPinModeInputPullupV
 - GpioPinMacros.h, [220](#)
- setGpioPinModeInputV
 - GpioPinMacros.h, [221](#)
- setGpioPinModeOutput
 - GpioPinMacros.h, [218](#)
- setGpioPinModeOutputV
 - GpioPinMacros.h, [221](#)
- setTimeout
 - Reader, [98](#)
 - Serial0, [122](#)
 - Serial1, [141](#)
 - Serial2, [160](#)
 - Serial3, [179](#)
- SimpleDelays.h, [263](#), [266](#)
 - delayQuartersOfMicroSeconds, [265](#)
 - delayTenthsOfSeconds, [265](#)
 - delayWholeMilliSeconds, [265](#)
- SPI, [51](#)
 - ByteOrder, [53](#)
 - configure, [54](#)
 - disable, [54](#)
 - enable, [54](#)
 - kLsbFirst, [53](#)
 - kMsbFirst, [53](#)
 - kSpiMode0, [53](#)
 - kSpiMode1, [53](#)
 - kSpiMode2, [53](#)
 - kSpiMode3, [53](#)
 - SpiMode, [53](#)
 - transmit, [55](#)
 - transmit16, [55](#)
 - transmit32, [56](#)
- SPI.h, [267](#), [268](#)
- SPI::SPISettings, [182](#)
 - getSpcr, [183](#)
 - getSpsr, [183](#)
 - SPISettings, [182](#)
- SpiMode
 - SPI, [53](#)
- SPISettings
 - SPI::SPISettings, [182](#)
- start
 - I2cMaster, [39](#)
 - I2cSlave, [47](#)
 - Serial0, [122](#)
 - Serial1, [141](#)
 - Serial2, [160](#)
 - Serial3, [179](#)
 - USART0, [58](#)
 - USART1, [61](#)
 - USART2, [65](#)
 - USART3, [69](#)

- stop
 - I2cMaster, [39](#)
 - I2cSlave, [47](#)
 - Serial0, [123](#)
 - Serial1, [142](#)
 - Serial2, [161](#)
 - Serial3, [180](#)
 - USART0, [58](#)
 - USART1, [62](#)
 - USART2, [66](#)
 - USART3, [70](#)
- SystemClock.h, [272](#), [275](#)
 - delay, [273](#)
 - delayMicroseconds, [273](#)
 - delayMilliseconds, [274](#)
 - initSystemClock, [274](#)
 - micros, [274](#)
 - millis, [274](#)
- transmit
 - SPI, [55](#)
- transmit16
 - SPI, [55](#)
- transmit32
 - SPI, [56](#)
- transmitUSART0
 - USART0Minimal.h, [282](#)
- transmitUSART1
 - USART1Minimal.h, [290](#), [291](#)
- transmitUSART2
 - USART2Minimal.h, [299](#), [300](#)
- transmitUSART3
 - USART3Minimal.h, [308](#), [309](#)
- USART0, [56](#)
 - available, [57](#)
 - flush, [57](#)
 - peek, [57](#)
 - read, [58](#)
 - start, [58](#)
 - stop, [58](#)
 - write, [58](#), [59](#)
- USART0.h, [276](#), [279](#)
 - kSerial_5E1, [278](#)
 - kSerial_5E2, [278](#)
 - kSerial_5N1, [278](#)
 - kSerial_5N2, [278](#)
 - kSerial_5O1, [278](#)
 - kSerial_5O2, [278](#)
 - kSerial_6E1, [278](#)
 - kSerial_6E2, [278](#)
 - kSerial_6N1, [278](#)
 - kSerial_6N2, [278](#)
 - kSerial_6O1, [278](#)
 - kSerial_6O2, [278](#)
- kSerial_7E1, [278](#)
- kSerial_7E2, [278](#)
- kSerial_7N1, [278](#)
- kSerial_7N2, [278](#)
- kSerial_7O1, [278](#)
- kSerial_7O2, [278](#)
- kSerial_8E1, [278](#)
- kSerial_8E2, [278](#)
- kSerial_8N1, [278](#)
- kSerial_8N2, [278](#)
- kSerial_8O1, [278](#)
- kSerial_8O2, [278](#)
- UsartSerialConfiguration, [278](#)
- USART0Minimal.h, [281](#), [283](#)
 - initUSART0, [282](#)
 - receiveUSART0, [282](#)
 - releaseUSART0, [282](#)
 - transmitUSART0, [282](#)
- USART1, [60](#)
 - available, [61](#)
 - flush, [61](#)
 - peek, [61](#)
 - read, [61](#)
 - start, [61](#)
 - stop, [62](#)
 - write, [62](#), [63](#)
- USART1.h, [284](#), [287](#)
 - kSerial_5E1, [286](#)
 - kSerial_5E2, [286](#)
 - kSerial_5N1, [286](#)
 - kSerial_5N2, [286](#)
 - kSerial_5O1, [286](#)
 - kSerial_5O2, [286](#)
 - kSerial_6E1, [286](#)
 - kSerial_6E2, [286](#)
 - kSerial_6N1, [286](#)
 - kSerial_6N2, [286](#)
 - kSerial_6O1, [286](#)
 - kSerial_6O2, [286](#)
 - kSerial_7E1, [286](#)
 - kSerial_7E2, [286](#)
 - kSerial_7N1, [286](#)
 - kSerial_7N2, [286](#)
 - kSerial_7O1, [286](#)
 - kSerial_7O2, [286](#)
 - kSerial_8E1, [286](#)
 - kSerial_8E2, [286](#)
 - kSerial_8N1, [286](#)
 - kSerial_8N2, [286](#)
 - kSerial_8O1, [286](#)
 - kSerial_8O2, [286](#)
 - UsartSerialConfiguration, [286](#)
- USART1Minimal.h, [289](#), [292](#)
 - initUSART1, [290](#)

- receiveUSART1, [290](#)
- releaseUSART1, [290](#)
- transmitUSART1, [290](#), [291](#)
- USART2, [64](#)
 - available, [65](#)
 - flush, [65](#)
 - peek, [65](#)
 - read, [65](#)
 - start, [65](#)
 - stop, [66](#)
 - write, [66](#), [67](#)
- USART2.h, [292](#), [296](#)
 - kSerial_5E1, [295](#)
 - kSerial_5E2, [295](#)
 - kSerial_5N1, [295](#)
 - kSerial_5N2, [295](#)
 - kSerial_5O1, [295](#)
 - kSerial_5O2, [295](#)
 - kSerial_6E1, [295](#)
 - kSerial_6E2, [295](#)
 - kSerial_6N1, [295](#)
 - kSerial_6N2, [295](#)
 - kSerial_6O1, [295](#)
 - kSerial_6O2, [295](#)
 - kSerial_7E1, [295](#)
 - kSerial_7E2, [295](#)
 - kSerial_7N1, [295](#)
 - kSerial_7N2, [295](#)
 - kSerial_7O1, [295](#)
 - kSerial_7O2, [295](#)
 - kSerial_8E1, [295](#)
 - kSerial_8E2, [295](#)
 - kSerial_8N1, [295](#)
 - kSerial_8N2, [295](#)
 - kSerial_8O1, [295](#)
 - kSerial_8O2, [295](#)
 - UsartSerialConfiguration, [295](#)
- USART2Minimal.h, [298](#), [301](#)
 - initUSART2, [299](#)
 - receiveUSART2, [299](#)
 - releaseUSART2, [299](#)
 - transmitUSART2, [299](#), [300](#)
- USART3, [68](#)
 - available, [69](#)
 - flush, [69](#)
 - peek, [69](#)
 - read, [69](#)
 - start, [69](#)
 - stop, [70](#)
 - write, [70](#), [71](#)
- USART3.h, [301](#), [305](#)
 - kSerial_5E1, [304](#)
 - kSerial_5E2, [304](#)
 - kSerial_5N1, [304](#)
 - kSerial_5N2, [304](#)
 - kSerial_5O1, [304](#)
 - kSerial_5O2, [304](#)
 - kSerial_6E1, [304](#)
 - kSerial_6E2, [304](#)
 - kSerial_6N1, [304](#)
 - kSerial_6N2, [304](#)
 - kSerial_6O1, [304](#)
 - kSerial_6O2, [304](#)
 - kSerial_7E1, [304](#)
 - kSerial_7E2, [304](#)
 - kSerial_7N1, [304](#)
 - kSerial_7N2, [304](#)
 - kSerial_7O1, [304](#)
 - kSerial_7O2, [304](#)
 - kSerial_8E1, [304](#)
 - kSerial_8E2, [304](#)
 - kSerial_8N1, [304](#)
 - kSerial_8N2, [304](#)
 - kSerial_8O1, [304](#)
 - kSerial_8O2, [304](#)
 - UsartSerialConfiguration, [304](#)
- USART3Minimal.h, [307](#), [310](#)
 - initUSART3, [308](#)
 - receiveUSART3, [308](#)
 - releaseUSART3, [308](#)
 - transmitUSART3, [308](#), [309](#)
- UsartSerialConfiguration
 - USART0.h, [278](#)
 - USART1.h, [286](#)
 - USART2.h, [295](#)
 - USART3.h, [304](#)
- write
 - I2cLcd, [89](#), [90](#)
 - Serial0, [123](#), [124](#)
 - Serial1, [142](#), [143](#)
 - Serial2, [161](#), [162](#)
 - Serial3, [180](#), [181](#)
 - USART0, [58](#), [59](#)
 - USART1, [62](#), [63](#)
 - USART2, [66](#), [67](#)
 - USART3, [70](#), [71](#)
 - Writer, [193](#), [194](#)
- writeAsync
 - I2cMaster, [39–41](#)
- writeGpioPinDigital
 - GpioPinMacros.h, [219](#)
- writeGpioPinDigitalV
 - GpioPinMacros.h, [221](#)
- writeGpioPinPwm
 - Pwm.h, [249](#)
- writeGpioPinPwmV
 - Pwm.h, [253](#)

Writer, [183](#)
 IntegerOutputBase, [185](#)
 kBin, [185](#)
 kDec, [185](#)
 kHex, [185](#)
 kOct, [185](#)
 print, [186–189](#)
 println, [190–193](#)
 write, [193](#), [194](#)
Writer.h, [310](#), [311](#)
writeSync
 I2cMaster, [42](#), [43](#)