### AVRTools

Generated by Doxygen 1.8.11

# **Contents**

1	AVR	Tools:	A Library	for the AVR ATmega328 and ATmega2560 Microcontrollers	1
	1.1	Overvi	ew		1
	1.2	Audier	nce		2
	1.3	AVRT	ools is not.		2
	1.4	Quick	Tour of AV	'RTools	3
		1.4.1	Foundati	ional Elements and Concepts	3
		1.4.2	What you	u need to know about pin name macros	3
		1.4.3	The core	modules	5
			1.4.3.1	System initialization module	5
			1.4.3.2	System clock module	5
			1.4.3.3	Analog-to-Digital module	6
			1.4.3.4	PWM module	6
			1.4.3.5	Minimal USART modules	6
			1.4.3.6	ABI module	7
			1.4.3.7	New module	7
	1.5	Sampl	e start up	code using AVRTools	7
	1.6	Advan	ced modul	les	8
	1.7	Questi	ons		8

iv CONTENTS

2	Adva	vanced 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	13				
	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	17				
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: how can that work?	20				
	3.8	Why is there a setGpioPinHigh() macro and a setGpioPinHighV() function?	20				
4		espace Index	21				
	4.1	Namespace List	21				
5	Hiera	archical Index	23				
	5.1	Class Hierarchy	23				

CONTENTS

6	Clas	s Index			25
	6.1	Class I	List		25
7	File	Index			27
	7.1	File Lis	st		27
8	Nam	nespace	Documer	ntation	31
	8.1	I2cMas	ster Names	space Reference	31
		8.1.1	Detailed	Description	32
		8.1.2	Enumera	tion Type Documentation	33
			8.1.2.1	I2cBusSpeed	33
			8.1.2.2	I2cPullups	33
			8.1.2.3	I2cSendErrorCodes	33
			8.1.2.4	I2cStatusCodes	34
		8.1.3	Function	Documentation	34
			8.1.3.1	busy()	34
			8.1.3.2	pullups(uint8_t set=kPullupsOn)	34
			8.1.3.3	readAsync(uint8_t address, uint8_t numberBytes, volatile uint8_t *destination, volatile uint8_t *bytesRead, volatile uint8_t *status)	34
			8.1.3.4	readAsync(uint8_t address, uint8_t registerAddress, uint8_t numberBytes, volatile uint8_t *destination, volatile uint8_t *bytesRead, volatile uint8_t *status)	35
			8.1.3.5	readSync(uint8_t address, uint8_t numberBytes, uint8_t *destination)	35
			8.1.3.6	readSync(uint8_t address, uint8_t registerAddress, uint8_t numberBytes, uint8_⇔ t *destination)	36
			8.1.3.7	start(uint8_t speed=kl2cBusFast)	36
			8.1.3.8	stop()	36
			8.1.3.9	writeAsync(uint8_t address, uint8_t registerAddress, volatile uint8_t *status)	36
			8.1.3.10	writeAsync(uint8_t address, uint8_t registerAddress, uint8_t data, volatile uint8_← t ∗status)	37
			8.1.3.11	writeAsync(uint8_t address, uint8_t registerAddress, const char ∗data, volatile uint8← _t ∗status)	37

vi CONTENTS

		8.1.3.12	writeAsync(uint8_t address, uint8_t registerAddress, uint8_t *data, uint8_t number ← Bytes, volatile uint8_t *status)	38
		8.1.3.13	writeSync(uint8_t address, uint8_t registerAddress)	38
		8.1.3.14	writeSync(uint8_t address, uint8_t registerAddress, uint8_t data)	39
		8.1.3.15	writeSync(uint8_t address, uint8_t registerAddress, const char *data)	39
		8.1.3.16	writeSync(uint8_t address, uint8_t registerAddress, uint8_t *data, uint8_t numberBytes)	39
8.2	I2cSlav	ve Namesp	pace Reference	40
	8.2.1	Detailed	Description	40
	8.2.2	Enumera	ation Type Documentation	41
		8.2.2.1	I2cBusSpeed	41
		8.2.2.2	I2cPullups	41
		8.2.2.3	I2cStatusCodes	41
	8.2.3	Function	Documentation	41
		8.2.3.1	busy()	41
		8.2.3.2	processI2cMessage(uint8_t *buffer, uint8_t len)	42
		8.2.3.3	pullups(uint8_t set=kPullupsOn)	42
		8.2.3.4	start(uint8_t ownAddress, uint8_t speed=kl2cBusFast, bool answerGeneralCall=false)	43
		8.2.3.5	stop()	43
8.3	Interru	pts Names	space Reference	43
	8.3.1	Detailed	Description	43
	8.3.2	Enumera	tion Type Documentation	44
		8.3.2.1	ExternalInterrupts	44
		8.3.2.2	PinChangeInterrupts	44
8.4	MemU	tils Names	space Reference	44
	8.4.1	Detailed	Description	45
	8.4.2	Function	Documentation	45
		8.4.2.1	freeRam()	45
		8.4.2.2	freeRamQuickEstimate()	45

CONTENTS vii

8.5	SPI Na	amespace	nespace Reference		
	8.5.1	Detailed	Detailed Description		
	8.5.2	Enumera	tion Type Documentation	47	
		8.5.2.1	ByteOrder	47	
		8.5.2.2	SpiMode	47	
	8.5.3	Function	Documentation	48	
		8.5.3.1	configure(SPISettings settings)	48	
		8.5.3.2	disable()	48	
		8.5.3.3	enable()	49	
		8.5.3.4	transmit(uint8_t data)	49	
		8.5.3.5	transmit(uint8_t *buffer, size_t count)	49	
		8.5.3.6	transmit16(uint16_t data)	49	
		8.5.3.7	transmit32(uint32_t data)	50	
8.6	USAR	T0 Names	pace Reference	50	
	8.6.1	Detailed	Description	50	
	8.6.2	Function	Documentation	51	
		8.6.2.1	available()	51	
		8.6.2.2	flush()	51	
		8.6.2.3	peek()	51	
		8.6.2.4	read()	51	
		8.6.2.5	start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1)	51	
		8.6.2.6	stop()	52	
		8.6.2.7	write(char c)	52	
		8.6.2.8	write(const char *c)	52	
		8.6.2.9	write(const char *c, size_t n)	53	
		8.6.2.10	write(const uint8_t *c, size_t n)	53	
8.7	USAR	T1 Names	pace Reference	53	
	8.7.1	Detailed	Description	54	

viii CONTENTS

8.7.2	Function	Documentation	54
	8.7.2.1	available()	54
	8.7.2.2	flush()	54
	8.7.2.3	peek()	55
	8.7.2.4	read()	55
	8.7.2.5	start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1)	55
	8.7.2.6	stop()	55
	8.7.2.7	write(char c)	56
	8.7.2.8	write(const char *c)	56
	8.7.2.9	write(const char *c, size_t n)	56
	8.7.2.10	write(const uint8_t *c, size_t n)	57
USAR	Γ2 Names <sub> </sub>	pace Reference	57
8.8.1	Detailed	Description	58
8.8.2	Function	Documentation	58
	8.8.2.1	available()	58
	8.8.2.2	flush()	58
	8.8.2.3	peek()	58
	8.8.2.4	read()	58
	8.8.2.5	start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1)	59
	8.8.2.6	stop()	59
	8.8.2.7	write(char c)	59
	8.8.2.8	write(const char *c)	59
	8.8.2.9	write(const char *c, size_t n)	60
	8.8.2.10	write(const uint8_t *c, size_t n)	60
USAR	Γ3 Names <sub> </sub>	pace Reference	60
8.9.1	Detailed	Description	61
8.9.2	Function	Documentation	61
	8.9.2.1	available()	61
	8.9.2.2	flush()	61
	8.9.2.3	peek()	62
	8.9.2.4	read()	62
	8.9.2.5	start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1)	62
	8.9.2.6	stop()	62
	8.9.2.7	write(char c)	63
	8.9.2.8	write(const char *c)	63
	8.9.2.9	write(const char *c, size_t n)	63
	8.9.2.10	write(const uint8_t *c, size_t n)	64
	USAR <sup>-</sup> 8.8.1 8.8.2 USAR <sup>-</sup> 8.9.1	8.7.2.1 8.7.2.2 8.7.2.3 8.7.2.4 8.7.2.5 8.7.2.6 8.7.2.7 8.7.2.8 8.7.2.9 8.7.2.10  USART2 Names 8.8.1 Detailed 8.8.2 Function 8.8.2.1 8.8.2.2 8.8.2.3 8.8.2.4 8.8.2.5 8.8.2.6 8.8.2.7 8.8.2.8 8.8.2.9 8.8.2.10  USART3 Names 8.9.1 Detailed 8.9.2 Function 8.9.2.1 8.9.2.2 8.9.2.3 8.9.2.4 8.9.2.5 8.9.2.6 8.9.2.7 8.9.2.8 8.9.2.9	8.7.2.1 available() 8.7.2.2 flush() 8.7.2.3 peek() 8.7.2.4 read() 8.7.2.5 start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 8.7.2.6 stop() 8.7.2.7 write(char c) 8.7.2.9 write(const char *c, size_t n) 8.7.2.10 write(const uint8_t *c, size_t n) 8.7.2.10 write(const uint8_t *c, size_t n) 8.8.2.1 varialable() 8.8.2.2 flush() 8.8.2.2 flush() 8.8.2.3 peek() 8.8.2.4 read() 8.8.2.5 start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 8.8.2.6 stop() 8.8.2.7 write(char c) 8.8.2.8 write(const char *c, size_t n) 8.8.2.10 write(const uint8_t *c, size_t n)  USART3 Namespace Reference 8.9.1 Detailed Description 8.9.2 Function Documentation 8.9.2.1 available() 8.9.2.2 flush() 8.9.2.3 peek() 8.9.2.4 read() 8.9.2.5 start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 8.9.2.1 available() 8.9.2.2 flush() 8.9.2.3 peek() 8.9.2.4 read() 8.9.2.5 start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 8.9.2.1 available() 8.9.2.2 flush() 8.9.2.3 peek() 8.9.2.4 read() 8.9.2.5 start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 8.9.2.7 write(const char *c, size_t n) 8.9.2.8 write(const char *c) 8.9.2.8 write(const char *c) 8.9.2.8 write(const char *c) 8.9.2.9 write(const char *c) 8.9.2.9 write(const char *c) 8.9.2.8 write(const char *c) 8.9.2.9 write(const char *c) 8.9.2.1 write(char c)

CONTENTS ix

9	Clas	ass Documentation 65				
	9.1	Interru	ots::AllOff	Class Reference	65	
		9.1.1	Detailed	Description	65	
	9.2	Interru	pts::Extern	alOff Class Reference	65	
		9.2.1	Detailed	Description	66	
		9.2.2	Construc	tor & Destructor Documentation	66	
			9.2.2.1	ExternalOff(uint8_t whichOnesToTurnOff=kExternalInterruptMask)	66	
	9.3	GpioPi	nVariable (	Class Reference	66	
		9.3.1	Detailed	Description	67	
	9.4	I2cLcd	Class Ref	erence	67	
		9.4.1	Detailed	Description	70	
		9.4.2	Member	Enumeration Documentation	70	
			9.4.2.1	anonymous enum	70	
			9.4.2.2	anonymous enum	71	
			9.4.2.3	IntegerOutputBase	71	
		9.4.3	Member	Function Documentation	71	
			9.4.3.1	command(uint8_t cmd)	71	
			9.4.3.2	displayBottomRow(const char *str)	71	
			9.4.3.3	displayTopRow(const char *str)	72	
			9.4.3.4	init()	72	
			9.4.3.5	print(const char *str, bool addLn=false)	72	
			9.4.3.6	print(const uint8_t *buf, size_t size, bool addLn=false)	72	
			9.4.3.7	print(char c, bool addLn=false)	73	
			9.4.3.8	print(int n, int base=kDec, bool addLn=false)	73	
			9.4.3.9	print(unsigned int n, int base=kDec, bool addLn=false)	73	
			9.4.3.10	print(long n, int base=kDec, bool addLn=false)	74	
			9.4.3.11	print(unsigned long n, int base=kDec, bool addLn=false)	74	
			9.4.3.12	print(double d, int digits=2, bool addLn=false)	74	

x CONTENTS

		9.4.3.13	println(const char *str)	75
		9.4.3.14	println(const uint8_t *buf, size_t size)	75
		9.4.3.15	println(char c)	75
		9.4.3.16	println(unsigned char n, int base=kDec)	75
		9.4.3.17	println(int n, int base=kDec)	76
		9.4.3.18	println(unsigned int n, int base=kDec)	76
		9.4.3.19	println(long n, int base=kDec)	76
		9.4.3.20	println(unsigned long n, int base=kDec)	77
		9.4.3.21	println(double d, int digits=2)	77
		9.4.3.22	readButtons()	77
		9.4.3.23	setBacklight(uint8_t color)	77
		9.4.3.24	setCursor(uint8_t row, uint8_t col)	78
		9.4.3.25	write(char c)	78
		9.4.3.26	write(const char *str)	78
		9.4.3.27	write(const char *buffer, size_t size)	78
		9.4.3.28	write(const uint8_t *buffer, size_t size)	79
9.5	Interru	pts::PinCh	angeOff Class Reference	79
	9.5.1	Detailed	Description	79
	9.5.2	Construc	tor & Destructor Documentation	80
		9.5.2.1	PinChangeOff(uint8_t whichOnesToTurnOff=kPinChangeInterruptMask)	80
9.6	Reade	r Class Re	eference	80
	9.6.1	Detailed	Description	82
	9.6.2	Member	Function Documentation	82
		9.6.2.1	available()=0	82
		9.6.2.2	find(const char *target)	82
		9.6.2.3	find(const char *target, size_t length)	83
		9.6.2.4	findUntil(const char *target, const char *terminator)	83
		9.6.2.5	findUntil(const char *target, size_t targetLen, const char *terminate, size_t termLen) .	83

CONTENTS xi

		9.6.2.6	peek()=0	84
		9.6.2.7	read()=0	84
		9.6.2.8	readBytes(char *buffer, size_t length)	84
		9.6.2.9	readBytes(uint8_t *buffer, size_t length)	84
		9.6.2.10	readBytesUntil(char terminator, char *buffer, size_t length)	85
		9.6.2.11	readBytesUntil(uint8_t terminator, uint8_t *buffer, size_t length)	85
		9.6.2.12	readFloat(float *result)	85
		9.6.2.13	readFloat(float *result, char skipChar)	86
		9.6.2.14	readLine(char *buffer, size_t length)	86
		9.6.2.15	readLong(long *result)	86
		9.6.2.16	readLong(long *result, char skipChar)	87
		9.6.2.17	setTimeout(unsigned long milliseconds)	87
9.7	RingBu	ıffer Class	Reference	87
	9.7.1	Detailed	Description	88
	9.7.2	Construc	tor & Destructor Documentation	88
		9.7.2.1	RingBuffer(unsigned char *buffer, unsigned short size)	88
	9.7.3	Member	Function Documentation	89
		9.7.3.1	isEmpty()	89
		9.7.3.2	isFull()	89
		9.7.3.3	isNotEmpty()	89
		9.7.3.4	isNotFull()	89
		9.7.3.5	peek(unsigned short index=0)	89
		9.7.3.6	pull()	90
		9.7.3.7	push(unsigned char element)	90
9.8	RingBu	ıfferT< T, I	N, SIZE > Class Template Reference	90
	9.8.1	Detailed	Description	91
	9.8.2	Member	Function Documentation	91
		9.8.2.1	discardFromFront(N nbrElements)	91

xii CONTENTS

		9.8.2.2	isEmpty()
		9.8.2.3	isFull()
		9.8.2.4	isNotEmpty()
		9.8.2.5	isNotFull()
		9.8.2.6	peek(N index=0)
		9.8.2.7	pull()
		9.8.2.8	push(T element)
9.9	Serial0	Class Re	ference
	9.9.1	Detailed	Description
	9.9.2	Member	Enumeration Documentation
		9.9.2.1	IntegerOutputBase
	9.9.3	Member	Function Documentation
		9.9.3.1	available()
		9.9.3.2	find(const char *target)
		9.9.3.3	find(const char *target, size_t length)
		9.9.3.4	findUntil(const char *target, const char *terminator)
		9.9.3.5	findUntil(const char *target, size_t targetLen, const char *terminate, size_t termLen) . 98
		9.9.3.6	peek()
		9.9.3.7	print(const char *str, bool addLn=false)
		9.9.3.8	print(const uint8_t *buf, size_t size, bool addLn=false)
		9.9.3.9	print(char c, bool addLn=false)
		9.9.3.10	print(int n, int base=kDec, bool addLn=false)
		9.9.3.11	print(unsigned int n, int base=kDec, bool addLn=false)
		9.9.3.12	print(long n, int base=kDec, bool addLn=false)
		9.9.3.13	print(unsigned long n, int base=kDec, bool addLn=false)
		9.9.3.14	print(double d, int digits=2, bool addLn=false)
		9.9.3.15	println(const char *str)
		9.9.3.16	println(const uint8_t *buf, size_t size)

CONTENTS xiii

	9.9.3.17	println(char c)
	9.9.3.18	println(unsigned char n, int base=kDec)
	9.9.3.19	println(int n, int base=kDec)
	9.9.3.20	println(unsigned int n, int base=kDec)
	9.9.3.21	println(long n, int base=kDec)
	9.9.3.22	println(unsigned long n, int base=kDec)
	9.9.3.23	println(double d, int digits=2)
	9.9.3.24	read()
	9.9.3.25	readBytes(char *buffer, size_t length)
	9.9.3.26	readBytes(uint8_t *buffer, size_t length)
	9.9.3.27	readBytesUntil(char terminator, char *buffer, size_t length)
	9.9.3.28	readBytesUntil(uint8_t terminator, uint8_t *buffer, size_t length)
	9.9.3.29	readFloat(float *result)
	9.9.3.30	readFloat(float *result, char skipChar)
	9.9.3.31	readLine(char *buffer, size_t length)
	9.9.3.32	readLong(long *result)
	9.9.3.33	readLong(long *result, char skipChar)
	9.9.3.34	setTimeout(unsigned long milliseconds)
	9.9.3.35	start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 107
	9.9.3.36	stop()
	9.9.3.37	write(char c)
	9.9.3.38	write(const char *str)
	9.9.3.39	write(const char *buffer, size_t size)
	9.9.3.40	write(const uint8_t *buffer, size_t size)
Serial1	Class Ref	rerence
9.10.1	Detailed I	Description
9.10.2	Member E	Enumeration Documentation
	9.10.2.1	IntegerOutputBase

9.10

xiv CONTENTS

9.10.3	Member F	Function Documentation
	9.10.3.1	available()
	9.10.3.2	find(const char *target)
	9.10.3.3	find(const char *target, size_t length)
	9.10.3.4	findUntil(const char *target, const char *terminator)
	9.10.3.5	findUntil(const char *target, size_t targetLen, const char *terminate, size_t termLen) .113
	9.10.3.6	peek()
	9.10.3.7	print(const char *str, bool addLn=false)
	9.10.3.8	print(const uint8_t *buf, size_t size, bool addLn=false)
	9.10.3.9	print(char c, bool addLn=false)
	9.10.3.10	print(int n, int base=kDec, bool addLn=false)
	9.10.3.11	print(unsigned int n, int base=kDec, bool addLn=false)
	9.10.3.12	print(long n, int base=kDec, bool addLn=false)
	9.10.3.13	print(unsigned long n, int base=kDec, bool addLn=false)
	9.10.3.14	print(double d, int digits=2, bool addLn=false)
	9.10.3.15	println(const char *str)
	9.10.3.16	println(const uint8_t *buf, size_t size)
	9.10.3.17	println(char c)
	9.10.3.18	println(unsigned char n, int base=kDec)
	9.10.3.19	println(int n, int base=kDec)
	9.10.3.20	println(unsigned int n, int base=kDec)
	9.10.3.21	println(long n, int base=kDec)
	9.10.3.22	println(unsigned long n, int base=kDec)
	9.10.3.23	println(double d, int digits=2)
	9.10.3.24	read()
	9.10.3.25	readBytes(char *buffer, size_t length)
	9.10.3.26	readBytes(uint8_t *buffer, size_t length)
	9.10.3.27	readBytesUntil(char terminator, char *buffer, size_t length)

CONTENTS xv

		9.10.3.28 readBytesUntil(uint8_t terminator, uint8_t *buffer, size_t length)
		9.10.3.29 readFloat(float *result)
		9.10.3.30 readFloat(float *result, char skipChar)
		9.10.3.31 readLine(char *buffer, size_t length)
		9.10.3.32 readLong(long *result)
		9.10.3.33 readLong(long *result, char skipChar)
		9.10.3.34 setTimeout(unsigned long milliseconds)
		9.10.3.35 start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 122
		9.10.3.36 stop()
		9.10.3.37 write(char c)
		9.10.3.38 write(const char *str)
		9.10.3.39 write(const char *buffer, size_t size)
		9.10.3.40 write(const uint8_t *buffer, size_t size)
9.11	Serial2	Class Reference
	9.11.1	Detailed Description
	9.11.2	Member Enumeration Documentation
		9.11.2.1 IntegerOutputBase
	9.11.3	Member Function Documentation
		9.11.3.1 available()
		9.11.3.2 find(const char *target)
		9.11.3.3 find(const char *target, size_t length)
		9.11.3.4 findUntil(const char *target, const char *terminator)
		9.11.3.5 findUntil(const char *target, size_t targetLen, const char *terminate, size_t termLen) . 129
		9.11.3.6 peek()
		9.11.3.7 print(const char *str, bool addLn=false)
		9.11.3.8 print(const uint8_t *buf, size_t size, bool addLn=false)
		9.11.3.9 print(char c, bool addLn=false)
		9.11.3.10 print(int n, int base=kDec, bool addLn=false)

xvi CONTENTS

9.11.3.11 print(unsigned int n, int base=kDec, bool addLn=false)
9.11.3.12 print(long n, int base=kDec, bool addLn=false)
9.11.3.13 print(unsigned long n, int base=kDec, bool addLn=false)
9.11.3.14 print(double d, int digits=2, bool addLn=false)
9.11.3.15 println(const char *str)
9.11.3.16 println(const uint8_t *buf, size_t size)
9.11.3.17 println(char c)
9.11.3.18 println(unsigned char n, int base=kDec)
9.11.3.19 println(int n, int base=kDec)
9.11.3.20 println(unsigned int n, int base=kDec)
9.11.3.21 println(long n, int base=kDec)
9.11.3.22 println(unsigned long n, int base=kDec)
9.11.3.23 println(double d, int digits=2)
9.11.3.24 read()
9.11.3.25 readBytes(char *buffer, size_t length)
9.11.3.26 readBytes(uint8_t *buffer, size_t length)
9.11.3.27 readBytesUntil(char terminator, char *buffer, size_t length)
9.11.3.28 readBytesUntil(uint8_t terminator, uint8_t *buffer, size_t length)
9.11.3.29 readFloat(float *result)
9.11.3.30 readFloat(float *result, char skipChar)
9.11.3.31 readLine(char *buffer, size_t length)
9.11.3.32 readLong(long *result)
9.11.3.33 readLong(long *result, char skipChar)
9.11.3.34 setTimeout(unsigned long milliseconds)
9.11.3.35 start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 138
9.11.3.36 stop()
9.11.3.37 write(char c)
9.11.3.38 write(const char *str)

CONTENTS xvii

	9.11.3.39 write(const char *buffer, size_t size)
	9.11.3.40 write(const uint8_t *buffer, size_t size)
9.12 Serial3	3 Class Reference
9.12.1	Detailed Description
9.12.2	Member Enumeration Documentation
	9.12.2.1 IntegerOutputBase
9.12.3	Member Function Documentation
	9.12.3.1 available()
	9.12.3.2 find(const char *target)
	9.12.3.3 find(const char *target, size_t length)
	9.12.3.4 findUntil(const char *target, const char *terminator)
	9.12.3.5 findUntil(const char *target, size_t targetLen, const char *terminate, size_t termLen) .144
	9.12.3.6 peek()
	9.12.3.7 print(const char *str, bool addLn=false)
	9.12.3.8 print(const uint8_t *buf, size_t size, bool addLn=false)
	9.12.3.9 print(char c, bool addLn=false)
	9.12.3.10 print(int n, int base=kDec, bool addLn=false)
	9.12.3.11 print(unsigned int n, int base=kDec, bool addLn=false)
	9.12.3.12 print(long n, int base=kDec, bool addLn=false)
	9.12.3.13 print(unsigned long n, int base=kDec, bool addLn=false)
	9.12.3.14 print(double d, int digits=2, bool addLn=false)
	9.12.3.15 println(const char *str)
	9.12.3.16 println(const uint8_t *buf, size_t size)
	9.12.3.17 println(char c)
	9.12.3.18 println(unsigned char n, int base=kDec)
	9.12.3.19 println(int n, int base=kDec)
	9.12.3.20 println(unsigned int n, int base=kDec)
	9.12.3.21 println(long n, int base=kDec)

xviii CONTENTS

		9.12.3.22 println(unsigned long n, int base=kDec)
		9.12.3.23 println(double d, int digits=2)
		9.12.3.24 read()
		9.12.3.25 readBytes(char *buffer, size_t length)
		9.12.3.26 readBytes(uint8_t *buffer, size_t length)
		9.12.3.27 readBytesUntil(char terminator, char *buffer, size_t length)
		9.12.3.28 readBytesUntil(uint8_t terminator, uint8_t *buffer, size_t length)
		9.12.3.29 readFloat(float *result)
		9.12.3.30 readFloat(float *result, char skipChar)
		9.12.3.31 readLine(char *buffer, size_t length)
		9.12.3.32 readLong(long *result)
		9.12.3.33 readLong(long *result, char skipChar)
		9.12.3.34 setTimeout(unsigned long milliseconds)
		9.12.3.35 start(unsigned long baudRate, UsartSerialConfiguration config=kSerial_8N1) 153
		9.12.3.36 stop()
		9.12.3.37 write(char c)
		9.12.3.38 write(const char *str)
		9.12.3.39 write(const char *buffer, size_t size)
		9.12.3.40 write(const uint8_t *buffer, size_t size)
9.13	SPI::SF	PISettings Class Reference
	9.13.1	Detailed Description
	9.13.2	Constructor & Destructor Documentation
		9.13.2.1 SPISettings(uint32_t maxSpeed, uint8_t bitOrder, uint8_t dataMode)
	9.13.3	Member Function Documentation
		9.13.3.1 getSpcr() const
		9.13.3.2 getSpsr() const
9.14	Writer (	Class Reference
	9.14.1	Detailed Description

CONTENTS xix

9.14.2	Member E	Enumeration Documentation
	9.14.2.1	IntegerOutputBase
9.14.3	Member F	Function Documentation
	9.14.3.1	print(const char *str, bool addLn=false)
	9.14.3.2	print(const uint8_t *buf, size_t size, bool addLn=false)
	9.14.3.3	print(char c, bool addLn=false)
	9.14.3.4	print(int n, int base=kDec, bool addLn=false)
	9.14.3.5	print(unsigned int n, int base=kDec, bool addLn=false)
	9.14.3.6	print(long n, int base=kDec, bool addLn=false)
	9.14.3.7	print(unsigned long n, int base=kDec, bool addLn=false)
	9.14.3.8	print(double d, int digits=2, bool addLn=false)
	9.14.3.9	println(const char *str)
	9.14.3.10	println(const uint8_t *buf, size_t size)
	9.14.3.11	println(char c)
	9.14.3.12	println(unsigned char n, int base=kDec)
	9.14.3.13	println(int n, int base=kDec)
	9.14.3.14	println(unsigned int n, int base=kDec)
	9.14.3.15	println(long n, int base=kDec)
	9.14.3.16	println(unsigned long n, int base=kDec)
	9.14.3.17	println(double d, int digits=2)
	9.14.3.18	write(char c)=0
	9.14.3.19	write(const char *str)=0
	9.14.3.20	write(const char *buffer, size_t size)=0
	9.14.3.21	write(const uint8_t *buffer, size_t size)=0

XX CONTENTS

10	File I	Docume	entation 16	<b>3</b> 7
	10.1	abi.h Fi	le Reference	37
		10.1.1	Detailed Description	37
	10.2	Analog	2Digital.h File Reference	38
		10.2.1	Detailed Description	39
		10.2.2	Macro Definition Documentation	39
			10.2.2.1 readGpioPinAnalog	39
		10.2.3	Enumeration Type Documentation	39
			10.2.3.1 A2DVoltageReference	39
		10.2.4	Function Documentation	70
			10.2.4.1 initA2D(uint8_t ref=kA2dReferenceAVCC)	70
			10.2.4.2 readA2D(int8_t channel)	70
			10.2.4.3 readGpioPinAnalogV(const GpioPinVariable &pinVar)	70
			10.2.4.4 setA2DVoltageReference(A2DVoltageReference ref)	71
			10.2.4.5 setA2DVoltageReference11V()	71
			10.2.4.6 setA2DVoltageReference256V()	71
			10.2.4.7 setA2DVoltageReferenceAREF()	71
			10.2.4.8 setA2DVoltageReferenceAVCC()	71
	10.3	Arduino	MegaPins.h File Reference	72
		10.3.1	Detailed Description	72
	10.4	Arduino	Pins.h File Reference	73
		10.4.1	Detailed Description	73
	10.5	Arduino	oUnoPins.h File Reference	74
		10.5.1	Detailed Description	74
	10.6	GpioPi	nMacros.h File Reference	74
		10.6.1	Detailed Description	77
		10.6.2	Macro Definition Documentation	77
			10.6.2.1 getGpioADC	77

CONTENTS xxi

	10.6.2.2 getGpioCOM
	10.6.2.3 getGpioDDR
	10.6.2.4 getGpioMASK
	10.6.2.5 getGpioOCR
	10.6.2.6 getGpioPIN
	10.6.2.7 getGpioPORT
	10.6.2.8 getGpioTCCR
	10.6.2.9 GpioPin
	10.6.2.10 GpioPinAnalog
	10.6.2.11 GpioPinPwm
	10.6.2.12 isGpioPinModeInput
	10.6.2.13 isGpioPinModeOutput
	10.6.2.14 makeGpioVarFromGpioPin
	10.6.2.15 makeGpioVarFromGpioPinAnalog
	10.6.2.16 makeGpioVarFromGpioPinPwm
	10.6.2.17 readGpioPinDigital
	10.6.2.18 setGpioPinHigh
	10.6.2.19 setGpioPinLow
	10.6.2.20 setGpioPinModeInput
	10.6.2.21 setGpioPinModeInputPullup
	10.6.2.22 setGpioPinModeOutput
	10.6.2.23 writeGpioPinDigital
10.6.3	Enumeration Type Documentation
	10.6.3.1 anonymous enum
10.6.4	Function Documentation
	10.6.4.1 isGpioPinModeInputV(const GpioPinVariable &pinVar)
	10.6.4.2 isGpioPinModeOutputV(const GpioPinVariable &pinVar)
	10.6.4.3 readGpioPinDigitalV(const GpioPinVariable &pinVar)

xxii CONTENTS

10.6.4.4 setGpioPinHighV(const GpioPinVariable &pinVar)
10.6.4.5 setGpioPinLowV(const GpioPinVariable &pinVar)
10.6.4.6 setGpioPinModeInputPullupV(const GpioPinVariable &pinVar)
10.6.4.7 setGpioPinModeInputV(const GpioPinVariable &pinVar)
10.6.4.8 setGpioPinModeOutputV(const GpioPinVariable &pinVar)
10.6.4.9 writeGpioPinDigitalV(const GpioPinVariable &pinVar, bool value)
10.7 I2cLcd.h File Reference
10.7.1 Detailed Description
10.8 I2cMaster.h File Reference
10.8.1 Detailed Description
10.9 I2cSlave.h File Reference
10.9.1 Detailed Description
10.10InitSystem.h File Reference
10.10.1 Detailed Description
10.10.2 Function Documentation
10.10.2.1 initSystem()
10.11 InterruptUtils.h File Reference
10.11.1 Detailed Description
10.12MemUtils.h File Reference
10.12.1 Detailed Description
10.13 new.h File Reference
10.13.1 Detailed Description
10.14Pwm.h File Reference
10.14.1 Detailed Description
10.14.2 Macro Definition Documentation
10.14.2.1 writeGpioPinPwm
10.14.3 Function Documentation
10.14.3.1 clearTimer0()

CONTENTS xxiii

10.14.3.2 clearTimer1()
10.14.3.3 clearTimer2()
10.14.3.4 clearTimer3()
10.14.3.5 clearTimer4()
10.14.3.6 clearTimer5()
10.14.3.7 initPwmTimer0()
10.14.3.8 initPwmTimer1()
10.14.3.9 initPwmTimer2()
10.14.3.1ünitPwmTimer3()
10.14.3.11initPwmTimer4()
10.14.3.12initPwmTimer5()
10.14.3.13writeGpioPinPwmV(const GpioPinVariable &pinVar, uint8_t value)
10.15Reader.h File Reference
10.15.1 Detailed Description
10.16RingBuffer.h File Reference
10.16.1 Detailed Description
10.17RingBufferT.h File Reference
10.17.1 Detailed Description
10.18SimpleDelays.h File Reference
10.18.1 Detailed Description
10.18.2 Function Documentation
10.18.2.1 delayQuartersOfMicroSeconds(uint16_t nbrOfQuartersOfMicroSeconds) 203
10.18.2.2 delayTenthsOfSeconds(uint8_t nbrOfTenthsOfSeconds)
10.18.2.3 delayWholeMilliSeconds(uint8_t nbrOfMilliSeconds)
10.19SPI.h File Reference
10.19.1 Detailed Description
10.20SystemClock.h File Reference
10.20.1 Detailed Description

xxiv CONTENTS

10.20.2 Function Documentation
10.20.2.1 delay(unsigned long ms)
10.20.2.2 delayMicroseconds(unsigned int us)
10.20.2.3 delayMilliseconds(unsigned long ms)
10.20.2.4 initSystemClock()
10.20.2.5 micros()
10.20.2.6 millis()
10.21USART0.h File Reference
10.21.1 Detailed Description
10.21.2 Enumeration Type Documentation
10.21.2.1 UsartSerialConfiguration
10.22USART0Minimal.h File Reference
10.22.1 Detailed Description
10.22.2 Function Documentation
10.22.2.1 initUSART0(unsigned long baudRate)
10.22.2.2 receiveUSART0()
10.22.2.3 releaseUSART0()
10.22.2.4 transmitUSART0(unsigned char data)
10.22.2.5 transmitUSART0(const char *data)
10.23USART1.h File Reference
10.23.1 Detailed Description
10.23.2 Enumeration Type Documentation
10.23.2.1 UsartSerialConfiguration
10.24USART1Minimal.h File Reference
10.24.1 Detailed Description
10.24.2 Function Documentation
10.24.2.1 initUSART1(unsigned long baudRate)
10.24.2.2 receiveUSART1()

CONTENTS xxv

10.24.2.3 releaseUSART1()	21
10.24.2.4 transmitUSART1(unsigned char data)	21
10.24.2.5 transmitUSART1(const char *data)	21
10.25USART2.h File Reference	22
10.25.1 Detailed Description	23
10.25.2 Enumeration Type Documentation	24
10.25.2.1 UsartSerialConfiguration	24
10.26USART2Minimal.h File Reference	26
10.26.1 Detailed Description	27
10.26.2 Function Documentation	27
10.26.2.1 initUSART2(unsigned long baudRate)	27
10.26.2.2 receiveUSART2()	27
10.26.2.3 releaseUSART2()	28
10.26.2.4 transmitUSART2(unsigned char data)	28
10.26.2.5 transmitUSART2(const char *data)	28
10.27USART3.h File Reference	29
10.27.1 Detailed Description	30
10.27.2 Enumeration Type Documentation	31
10.27.2.1 UsartSerialConfiguration	31
10.28USART3Minimal.h File Reference	3
10.28.1 Detailed Description	34
10.28.2 Function Documentation	34
10.28.2.1 initUSART3(unsigned long baudRate)	}4
10.28.2.2 receiveUSART3()	}4
10.28.2.3 releaseUSART3()	}5
10.28.2.4 transmitUSART3(unsigned char data)	}5
10.28.2.5 transmitUSART3(const char *data)	}5
10.29Writer.h File Reference	36
10.29.1 Detailed Description	36
Index 23	37

## **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 Ardiuno library interface while embracing the fundametal C/C++ philosopy 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 primary 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 he or she remembered to turn off PWM mode before using the pin digitally. Assuming the programmer knows what he or she is 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 1 cycle (it's actually a macro in AVRTools).

#### 1.2 Audience

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

• 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 single-cycle in and out 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 or 16 MHz (the code automatically adapts to these two clock speeds). I have not tested any other clock speeds, and some of the delay functions are specifically coded for 8 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 USARTO 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 3

#### 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 the analog channel associated with the pin. For pins that support PWM, the single name also provides access to 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 single-cycle in, out, sbi, cbi, sbic, or sbis instructions for such operations. 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 labelled pins on the Arduino boards. You can use these pin names to define your own macros:

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 bit
- setGpioPinModeInputPullup( pin ) Clear the corresponding DDRn and PORTn bits
- 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 );
}</pre>
```

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.
- 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 Quick Tour of AVRTools 5

#### 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 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 delay( unsigned long ms );
```

Note that unlike the Arduino libary, 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( avrPort );  /* 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( avrPort, 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 <code>initPwmTimerN()</code> 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, 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. 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"
init 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 );
       delay( 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 surpressing 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.

#### **Documentation**

Detailed documentation is provided by this PDF document located in the repository, or online in HTML form.

#### 1.7 Questions

If you have questions, please check out the FAQ.

# **Chapter 2**

# **Advanced Features**

The AVRTools library includes either more advanced 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 core modules and, sometimes, 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 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). The receive buffer, however, will overwrite if it gets full. You must clear the

10 Advanced Features

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 are provided. The first is provided in namespace USART0 and 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. 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 or USART0 requires linking against SystemClock.cpp and calling 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

These two modules provide two different interfaces to the two-wire serial interface (TWI) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. There are two different modules corresponding to whether your application will function as a Master (as per the I2C protocol), or as a Slave.

#### 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 which 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.

2.2 I2C modules 11

#### 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 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.

12 Advanced Features

## 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 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 assocaited 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 desireable. 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 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:

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 13

## 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 an interrupt functions (analogous to the I2C hardware), AVRTools does not implement asynchronous SPI transmission, instead implementing sychronous 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 aready 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 InterruptUtils 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:

14 Advanced Features

# 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. The primary function is freeRam() which returns the number of free bytes remaining in SRAM.

# 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. These 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.

2.8 GPIO pin variables

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. Access 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 also three macros that you can use to initialize GPIO pin variables of type GpioPinVariable. The three are: 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 );
```

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
isGpioPinModeOutput( pinMacro )	isGpioPinModeOutputV( const	Is the corresponding DDRn bit set?
	GpioPinVariable& pinVar )	
isGpioPinModeInput( pinMacro )	isGpioPinModeInputV( const Gpio↔	Is the corresponding DDRn bit clear?
	PinVariable& pinVar )	
setGpioPinModeOutput( pinMacro )	setGpioPinModeOutputV( const	Enable the corresponding DDRn bit
	GpioPinVariable& pinVar )	
setGpioPinModeInput( pinMacro )	setGpioPinModeInputV( const	Clear the corresponding DDRn bit
	GpioPinVariable& pinVar )	
setGpioPinModeInputPullup( pin←	setGpioPinModeInputPullupV( const	Clear the corresponding DDRn and
Macro)	GpioPinVariable& pinVar )	PORTn bits
readGpioPinDigital( pinMacro )	readGpioPinDigitalV( const Gpio↔	Return the value (0 or 1 ) of the cor-
	PinVariable& pinVar )	responding PINn bit
writeGpioPinDigital( pinMacro, value	writeGpioPinDigitalV( const Gpio↔	Write a 0 or 1 to the corresponding
)	PinVariable& pinVar, bool value)	PORTn bit

16 Advanced Features

Macro Version	Function Version	Purpose
setGpioPinHigh( pinMacro )	setGpioPinHighV( const GpioPin⊷	Set the corresponding PORTn bit
	Variable& pinVar )	
setGpioPinLow( pinMacro )	setGpioPinLowV( const GpioPin←	Clear the corresponding PORTn bit
	Variable& pinVar )	
readGpioPinAnalog( pinMacro )	readGpioPinAnalogV( const Gpio←	Read an analog value from the cor-
	PinVariable& pinVar )	responding ADC channel
writeGpioPinPwm( pinMacro, value )	writeGpioPinPwmV( const Gpio←	Set the corresonding PWM output
	PinVariable& pinVar, uint8_t value)	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
#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 GPIO pin variables

# 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"
                         pPin10
#define pRed
                       pPin07
pPin04
#define pYellow
#define pGreen
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++] );
        setGpioPinHighV( pins[i] );
```

18 Advanced Features

# **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: 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.

20 FAQ

# 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. Tomaž Šolc has done the research and posted the results on his blog. Check it out (and check out the 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 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: how can that work?

Someone has been reading 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 (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, pPin01). 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:

I2cMaste	er en	
	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	31
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	40
Interrupts	S	
	This namespace bundles various utility classes designed to suppress selected interrupts using the RAII idiom	43
MemUtils		
	A namespace providing encapsulation for functions that report the available memory in SRAM	44
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	45
USART0	·	
	This namespace bundles a high-level buffered interface to the USART0 hardware. It provides logical cohesion and prevents namespace collisions	50
<b>USART1</b>		
	This namespace bundles a high-level buffered interface to the USART1 hardware. It provides logical cohesion and prevents namespace collisions	53
USART2		
	This namespace bundles a high-level buffered interface to the USART2 hardware. It provides logical cohesion and prevents namespace collisions	57
<b>USART3</b>		
	This namespace bundles a high-level buffered interface to the USART3 hardware. It provides logical cohesion and prevents namespace collisions	60

22 Namespace Index

# **Chapter 5**

# **Hierarchical Index**

# 5.1 Class Hierarchy

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

Interrupts::AIIOff	65
Interrupts::ExternalOff	65
GpioPinVariable	66
Interrupts::PinChangeOff	79
Reader	80
Serial0	93
Serial1	09
Serial2	24
Serial3	40
RingBuffer	87
RingBufferT< T, N, SIZE >	90
SPI::SPISettings	55
Writer	57
I2cLcd	67
Serial0	93
Serial1	09
Serial2	24
Serial3	40

24 Hierarchical Index

# **Chapter 6**

# **Class Index**

# 6.1 Class List

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

Interrupts	s::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	65
Interrupts	s::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	65
GpioPin\	/ariable	
	This class defines a type that can encode a GPIO pin as a variable. Read the section on GPIO Pin Variables to understand how to use this class	66
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	67
Interrupts	s::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	79
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)	80
RingBuffe	er	
	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 storgage of bytes, providing a single code base that is shared by all instances of this class	87
RingBuffe	erT< T, N, SIZE >	
	Template-based ring buffer class that can store different kinds of objects in buffers of whatever size is needed	90
Serial0		
	Provides a high-end interface to serial communications using USART0	93
Serial1	<b>5</b>	
	Provides a high-end interface to serial communications using USART1	109

26 Class Index

Serial2	
	Provides a high-end interface to serial communications using USART2
Serial3	
	Provides a high-end interface to serial communications using USART3
SPI::SPI	Settings
	A class that binds settings for configuring SPI transmissions
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)

# **Chapter 7**

# File Index

# 7.1 File List

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

abi.h		
•	nctions needed to complete the avr-gcc C++ ABI. You never need to	
	y need to link against abi.cpp if you encounter certain link errors $\ldots \ldots$	167
Analog2Digital.h		
This file provides functions the	hat access the analog-to-digital conversion capability of the ATmega328	
and ATmega2560 microcont	trollers	168
ArduinoMegaPins.h		
	rd Arduino Uno pin name macros. It may be included directly by user	
	only user code includes the file ArduinoPins.h, which in turn includes this	
file (when compiling for Ardu	iino Uno targets)	172
ArduinoPins.h		
•	hat users should include to access and use the pin name macros	173
ArduinoUnoPins.h		
	rd Arduino Uno pin name macros. It may be included directly by user	
	only user code includes the file ArduinoPins.h, which in turn includes this	
	iino Uno targets)	174
GpioPinMacros.h		
	y macros for naming and manipulating GPIO pin names	174
I2cLcd.h		
	provides a high-level interface to an LCD offering an I2C interface. The	
	s is HD44780U controlled LCD driven via an MCP23017 that offers an	
•	e available from Adafruit and SparkFun). To use this class you must also	404
	e I2C Master package from I2cMaster.h	184
I2cMaster.h	and industrian and the TMU (burn union populations of the Audulian	
•	nat interface to the TWI (two-wire serial interface) hardware of the Arduino	
, ,	ino Mega (ATmega2560), providing a high-level interface to I2C protocol	
	s file if you want your application will operate in Master mode as defined	10E
In the I2C protocol		100
	nat interface to the TWI (two-wire serial interface) hardware of the Arduino	
•	ino Mega (ATmega2560), providing a high-level interface to I2C protocol	
	is file if you want your application will operate in Slave mode as defined	
in the I2C protocol	s life if you want your application will operate in Slave fillode as defined	188

28 File Index

InitSyste	m.h	
	Include this file to use the functions that initialize the microcontroller to a known, basic state	190
Interrupt	Utils.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	190
MemUtils	s.h	
	This file provides functions that provide information on the available memory in SRAM $\dots \dots \dots$	191
new.h		
	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	192
Pwm.h		
	This file provides functions that access the PWM capability of the ATmega328 and ATmega2560 microcontrollers	193
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	200
RingBuff	er.h	
	This file provides an efficient ring buffer implementation for storing bytes	201
RingBuff		
	This file provides a very flexible, template-based ring buffer implementation	201
SimpleD	·	
	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	202
SPI.h		
	This file provides an interface to SPI subsystem available on the AVR ATMega328p (Arduino Uno) and ATMega2560 (Arduino Mega) microcontrollers	204
SystemC		
	Include this file to use the functions that instantiate and access a system clock that counts elapsed milliseconds	206
USART0		
	This file provides functions that offer high-level interfaces to USART0 hardware, which is available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560)	208
USART0	Minimal.h	
	This file provides functions that provide a minimalist interface to USART0 available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560)	212
USART1		
	This file provides functions that offer high-level interfaces to USART1 hardware, which is available on Arduino Mega (ATMega2560)	215
USART1	Minimal.h	
	This file provides functions that provide a minimalist interface to USART1 available on the Arduino Mega (ATmega2560)	219
USART2		
	This file provides functions that offer high-level interfaces to USART2 hardware, which is available on Arduino Mega (ATMega2560)	222
USART2	Minimal.h	
	This file provides functions that provide a minimalist interface to USART2 available on the Arduino Mega (ATmega2560)	226
USART3	l.h	
	This file provides functions that offer high-level interfaces to USART3 hardware, which is available on Arduino Mega (ATMega2560)	229

7.1 File List 29

USART3	Minimal.h	
	This file provides functions that provide a minimalist interface to USART3 available on the Arduino	
	Mega (ATmega2560)	233
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	236

30 File Index

# **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**

## **Functions**

void start (uint8 t speed=kl2cBusFast)

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 \*bytes←
 Read, 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 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 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 enum I2cMaster::I2cSendErrorCodes

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

### **Enumerator**

```
    kI2cNoError No error.
    kI2cErrTxBufferFull The transmit buffer is full (try again later)
    kI2cErrMsgTooLong The message is too long for the transmit buffer.
    kI2cErrNullStatusPtr The pointer to the status variable is null (need to provide a valid pointer)
    kI2cErrWriteWithoutData No data provided to send.
    kI2cErrReadWithoutStorage Performing a write+read, but no buffer provided to store the "read" data.
```

#### 8.1.2.4 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 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 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 uint8_t l2cMaster::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.

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 countered 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 commincations status of this message will be reported (volatile because the value will be updates 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 uint8\_t l2cMaster::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 onebyte 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 countered 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 commincations status of this message will be reported
  (volatile because the value will be updates 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 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 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 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 (kl2cBusFast).

8.1.3.8 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 uint8\_t l2cMaster::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 commincations status of this message will be reported (volatile because the value will be updates 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 uint8\_t l2cMaster::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 commincations status of this message will be reported
  (volatile because the value will be updates 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 uint8\_t l2cMaster::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 commincations status of this message will be reported
  (volatile because the value will be updates 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 uint8\_t l2cMaster::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 commincations status of this message will be reported
  (volatile because the value will be updates 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 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 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).

8.1.3.15 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).

8.1.3.16 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).

# 8.2 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**

#### **Functions**

uint8\_t processl2cMessage (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 processI2c Message() 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 enum I2cSlave::I2cBusSpeed

This enum lists I2C bus speed configurations.

#### Enumerator

kl2cBusSlow | I2C slow (standard) mode: 100 KHz.

kl2cBusFast I2C fast mode: 400 KHz.

## 8.2.2.2 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 enum I2cSlave::I2cStatusCodes

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

#### Enumerator

kl2cCompletedOk I2C communications completed with no error.

kl2cError I2C communications encountered an error.

kl2cTxPartial I2C Master terminated transmission before all data were sent.

kl2cRxOverflow Recieved a message larger than can be held in the receive buffer.

#### 8.2.3 Function Documentation

## 8.2.3.1 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 uint8\_t l2cSlave::processl2cMessage ( 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 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 void l2cSlave::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.

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 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**

## 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 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)
    kExternalInterruptAII All external interrupts.
```

#### 8.3.2.2 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.
    kPinChangeInterruptAII All pin change interrupts.
```

# 8.4 MemUtils Namespace Reference

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

## **Functions**

• unsigned int freeRam ()

Get the number of free bytes remaining in SRAM.

• unsigned int freeRamQuickEstimate ()

Get a quick estimate of the number of free bytes remaining in SRAM.

# 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 unsigned int MemUtils::freeRam ( )

Get the number of free bytes remaining in SRAM.

## Returns

The number of free bytes remaining in SRAM.

8.4.2.2 unsigned int MemUtils::freeRamQuickEstimate ( )

Get a quick estimate of the number of free bytes remaining in SRAM.

This provides a quicker, but perhaps slightly inaccurate, estimate of the amount of free memory.

## Returns

The number of free bytes remaining in SRAM.

# 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**

#### **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. 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 );
 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 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 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 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
```

# 8.5.3.2 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 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 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.

```
8.5.3.5 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.

```
8.5.3.6 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 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 USARTO 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 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 void USART0::flush ( )
```

Flush transmit buffer.

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

```
8.6.2.3 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 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 void USART0::start ( unsigned long baudRate, UsartSerialConfiguration 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 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 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 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.

 $\bullet \ \, {\mbox{$\rm 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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

- $\bullet\ _{\rm C}$  the byte array to write into the transmit buffer.
- ullet 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 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 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).

9.4 I2cLcd Class Reference 67

# 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 to understand how to use this class.

There are also three macros that you need to create vaiables 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:

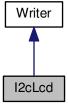
• GpioPinMacros.h

# 9.4 I2cLcd Class Reference

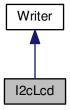
This class provides a high-level interface via I2C to an LCD such as those offered by AdaFruit and SparkFun. Specifically, it communicates via I2C with an MCP23017 that drives an HD44780U controlling an LCD. It also lets you detect button presses on the 5-button keypad generally assocaited with such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

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

Inheritance diagram for I2cLcd:



# Collaboration diagram for I2cLcd:



# **Public Types**

# **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.

9.4 I2cLcd Class Reference 69

· 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 (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 (unsigned char n, int base=kDec)

Print an unsigned character 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 assocaited 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

These constants are used to identify the five buttons.

# **Enumerator**

kButton\_Select the Select button
kButton\_Right the Right button
kButton\_Down the Down button
kButton\_Up the Up button
kButton\_Left the Left button

9.4 I2cLcd Class Reference 71

# 9.4.2.2 anonymous enum

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

### Enumerator

```
kBacklight_Red
kBacklight_Yellow
kBacklight_Green
kBacklight_Teal
Backlight teal.
kBacklight_Blue
kBacklight_Violet
kBacklight_White
Backlight white.
```

### **9.4.2.3 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 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 void I2cLcd::displayBottomRow ( const char \* str )

Display a C-string on the bottom row.

• str the C-string to display.

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

Display a C-string on the top row.

• str the C-string to display.

```
9.4.3.4 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 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.

```
9.4.3.6 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

9.4 I2cLcd Class Reference 73

```
9.4.3.7 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.

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

```
9.4.3.9 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

```
9.4.3.10 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.

```
9.4.3.11 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.

```
9.4.3.12 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

9.4 I2cLcd Class Reference 75

```
9.4.3.13 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.

```
9.4.3.14 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.

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

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

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

### Returns

```
9.4.3.17 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.

```
9.4.3.18 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.

```
9.4.3.19 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

9.4 I2cLcd Class Reference 77

```
9.4.3.20 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.

```
9.4.3.21 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 flaoting 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.

```
9.4.3.22 uint8_t l2cLcd::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  $k \leftarrow$  Button\_Right, kButton\_Left, kButton\_Down, kButton\_Up, or kButton\_Select to determine which buttons have been pressed.

```
9.4.3.23 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, k← Backlight\_Teal, kBacklight\_Blue, kBacklight\_Violet, or kBacklight\_White.

```
9.4.3.24 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).

```
9.4.3.25 size_t l2cLcd::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.

```
9.4.3.26 size_t l2cLcd::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.

```
9.4.3.27 size_t l2cLcd::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.

```
9.4.3.28 size_t l2cLcd::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

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:

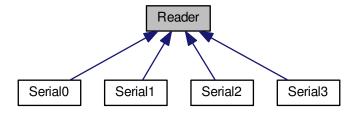
· InterruptUtils.h

# 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:



9.6 Reader Class Reference 81

### **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-termimated.

size\_t readBytesUntil (char terminator, char \*buffer, size\_t length)

Read characters from the input stream into a buffer, terminating when the terminator charactor 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 implementating 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 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 Serial1, Serial2, Serial3, and Serial0.

```
9.6.2.2 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 Reader Class Reference 83

```
9.6.2.3 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 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 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 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 Serial1, Serial2, Serial3, and Serial0.

```
9.6.2.7 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 Serial1, Serial2, Serial3, and Serial0.

```
9.6.2.8 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 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 Reader Class Reference 85

```
9.6.2.10 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 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 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 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 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 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 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 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 storgage 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 storgage of bytes, providing a single code base that is shared by all instances of this class.

For maximum flexiblity, the caller must provide the storage to be used for each RingBuffer object instantiated (this allows the use of different sized ring bufferss 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 ( 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 bool RingBuffer::isEmpty() [inline]
```

Determine if the buffer is empty.

#### Returns

true if the buffer is empty; false if not.

```
9.7.3.2 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 bool RingBuffer::isNotEmpty() [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 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 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 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 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**

	Т	is the type of object that will be stored in the RingBufferT instantiation.
	Ν	is the integer type that will be used to index the RingBufferT elements.
S	IZE	is an integer indicating the size of the RingBufferT instantiation.

## 9.8.2 Member Function Documentation

9.8.2.1 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 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 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 template < typename T , typename N , unsigned int SIZE> bool RingBufferT< T, N, SIZE >::isNotEmpty ( ) [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 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 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 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 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:

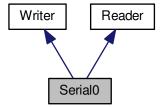
· RingBufferT.h

## 9.9 Serial Class Reference

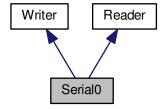
Provides a high-end interface to serial communications using USART0.

#include <USARTO.h>

Inheritance diagram for Serial0:



Collaboration diagram for Serial0:



## **Public Types**

## **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 (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 (unsigned char n, int base=kDec)

Print an unsigned character 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 charactor 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 USAR ← T0. 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 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 virtual bool SerialO::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 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 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 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 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 virtual int SerialO::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 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

```
9.9.3.8 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.

```
9.9.3.9 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.

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

```
9.9.3.11 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.

```
9.9.3.12 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.

```
9.9.3.13 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

```
9.9.3.14 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.

```
9.9.3.15 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.

```
9.9.3.16 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.

```
9.9.3.17 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

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

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

- n is the unsigned character 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.

```
9.9.3.19 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.

```
9.9.3.20 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

```
9.9.3.21 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.

```
9.9.3.22 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.

```
9.9.3.23 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 flaoting point number to output.
- digits is the number of decimal digits to output; the default is 2.

## Returns

```
9.9.3.24 virtual int SerialO::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.

```
9.9.3.25 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-termimated.

- 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).

```
9.9.3.26 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).

```
9.9.3.27 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 charactor 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).

```
9.9.3.28 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).

```
9.9.3.29 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.

```
9.9.3.30 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.

```
9.9.3.31 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-termimated.

- 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).

```
9.9.3.32 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.

```
9.9.3.33 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.

```
9.9.3.34 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.

```
9.9.3.35 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 SerialO on USARTO.

- 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.

```
9.9.3.36 void SerialO::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.

```
9.9.3.37 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 ).

• the character to be written.

### Returns

the number of bytes written.

Implements Writer.

```
9.9.3.38 virtual size_t SerialO::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.

```
9.9.3.39 virtual size_t SerialO::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.

```
9.9.3.40 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:

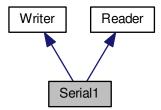
USART0.h

## 9.10 Serial1 Class Reference

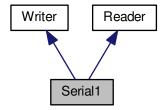
Provides a high-end interface to serial communications using USART1.

#include <USART1.h>

Inheritance diagram for Serial1:



Collaboration diagram for Serial1:



## **Public Types**

## **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 (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 (unsigned char n, int base=kDec)

Print an unsigned character 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-termimated.

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

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-termimated.

· 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 USAR ← T1. 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 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 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 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 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 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 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 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 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.

```
9.10.3.8 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.

```
9.10.3.9 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

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

```
9.10.3.11 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.

```
9.10.3.12 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

```
9.10.3.13 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.

```
9.10.3.14 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.

```
9.10.3.15 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

```
9.10.3.16 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.

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

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

- n is the unsigned character 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.

```
9.10.3.19 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

```
9.10.3.20 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.

```
9.10.3.21 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.

```
9.10.3.22 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

9.10 Serial1 Class Reference 119

```
9.10.3.23 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 flaoting 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.

```
9.10.3.24 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.

```
9.10.3.25 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).

```
9.10.3.26 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).

```
9.10.3.27 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).

```
9.10.3.28 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.
- $\bullet\,$  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).

```
9.10.3.29 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 Serial1 Class Reference 121

```
9.10.3.30 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.

```
9.10.3.31 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).

```
9.10.3.32 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.

```
9.10.3.33 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.

```
9.10.3.34 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.

```
9.10.3.35 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.

```
9.10.3.36 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 Serial1 Class Reference 123

```
9.10.3.37 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 ).

• the character to be written.

### Returns

the number of bytes written.

Implements Writer.

```
9.10.3.38 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.

```
9.10.3.39 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.

```
9.10.3.40 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:

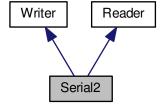
- USART1.h
- USART1.cpp

# 9.11 Serial2 Class Reference

Provides a high-end interface to serial communications using USART2.

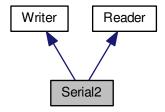
```
#include <USART2.h>
```

Inheritance diagram for Serial2:



9.11 Serial2 Class Reference 125

Collaboration diagram for Serial2:



# **Public Types**

### **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 (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 (unsigned char n, int base=kDec)

Print an unsigned character 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 charactor 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-termimated.

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 USAR ← T2. 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 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 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 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 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 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 Serial2 Class Reference 129

```
9.11.3.5 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 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 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

```
9.11.3.8 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.

```
9.11.3.9 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.

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

9.11 Serial2 Class Reference 131

```
9.11.3.11 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.

```
9.11.3.12 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.

```
9.11.3.13 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

```
9.11.3.14 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.

```
9.11.3.15 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.

```
9.11.3.16 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.

```
9.11.3.17 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

9.11 Serial2 Class Reference 133

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

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

- n is the unsigned character 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.

```
9.11.3.19 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.

```
9.11.3.20 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

```
9.11.3.21 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.

```
9.11.3.22 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.

```
9.11.3.23 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 flaoting point number to output.
- digits is the number of decimal digits to output; the default is 2.

# Returns

9.11 Serial2 Class Reference 135

```
9.11.3.24 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.

```
9.11.3.25 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).

```
9.11.3.26 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).

```
9.11.3.27 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 charactor 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).

```
9.11.3.28 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).

```
9.11.3.29 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.

```
9.11.3.30 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 Serial2 Class Reference 137

```
9.11.3.31 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).

```
9.11.3.32 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.

```
9.11.3.33 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.

```
9.11.3.34 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.

```
9.11.3.35 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.

```
9.11.3.36 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.

```
9.11.3.37 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 ).

• the character to be written.

## Returns

the number of bytes written.

Implements Writer.

9.11 Serial2 Class Reference 139

```
9.11.3.38 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.

```
9.11.3.39 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.

```
9.11.3.40 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:

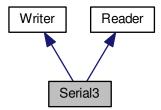
- USART2.h
- USART2.cpp

# 9.12 Serial3 Class Reference

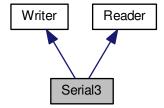
Provides a high-end interface to serial communications using USART3.

#include <USART3.h>

Inheritance diagram for Serial3:



Collaboration diagram for Serial3:



# **Public Types**

# **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 (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 (unsigned char n, int base=kDec)

Print an unsigned character 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-termimated.

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 charactor 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 USAR ← T3. 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 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)

## 9.12.3 Member Function Documentation

```
9.12.3.1 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 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 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 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 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 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 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.

```
9.12.3.8 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.

```
9.12.3.9 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.

- $_{\mbox{\scriptsize C}}$  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

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

```
9.12.3.11 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.

```
9.12.3.12 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

9.12 Serial3 Class Reference 147

```
9.12.3.13 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.

```
9.12.3.14 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.

```
9.12.3.15 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

```
9.12.3.16 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.

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

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

- n is the unsigned character 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.

```
9.12.3.19 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

9.12 Serial3 Class Reference 149

```
9.12.3.20 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.

```
9.12.3.21 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.

```
9.12.3.22 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

```
9.12.3.23 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 flaoting 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.

```
9.12.3.24 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.

```
9.12.3.25 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).

```
9.12.3.26 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 Serial3 Class Reference 151

```
9.12.3.27 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).

```
9.12.3.28 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).

```
9.12.3.29 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.

```
9.12.3.30 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.

```
9.12.3.31 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).

```
9.12.3.32 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 Serial3 Class Reference 153

```
9.12.3.33 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.

```
9.12.3.34 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.

```
9.12.3.35 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.

```
9.12.3.36 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.

```
9.12.3.37 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 ).

• the character to be written.

### Returns

the number of bytes written.

Implements Writer.

```
9.12.3.38 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.

```
9.12.3.39 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.

```
9.12.3.40 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.

156 Class Documentation

# 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 S← PISettings 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 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, kSpi

  Mode1, kSpiMode2, or kSpiMode3.

### 9.13.3 Member Function Documentation

```
9.13.3.1 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.

9.14 Writer Class Reference 157

```
9.13.3.2 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:

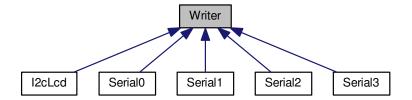
• SPI.h

# 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**

### **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.

158 Class Documentation

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 (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 (unsigned char n, int base=kDec)

Print an unsigned character 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 implementating the purely abstract functions in Writer.

9.14 Writer Class Reference 159

### 9.14.2 Member Enumeration Documentation

### 9.14.2.1 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 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.

```
9.14.3.2 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

160 Class Documentation

```
9.14.3.3 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.

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

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.

```
9.14.3.5 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

9.14 Writer Class Reference 161

```
9.14.3.6 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.

```
9.14.3.7 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.

```
9.14.3.8 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

162 Class Documentation

```
9.14.3.9 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.

```
9.14.3.10 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.

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

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

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

### Returns

9.14 Writer Class Reference 163

```
9.14.3.13 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.

```
9.14.3.14 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.

```
9.14.3.15 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

164 Class Documentation

```
9.14.3.16 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.

```
9.14.3.17 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 flaoting 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.

```
9.14.3.18 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 Serial1, Serial2, Serial3, Serial0, and I2cLcd.

9.14 Writer Class Reference 165

```
9.14.3.19 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 Serial1, Serial2, Serial3, Serial0, and I2cLcd.

```
9.14.3.20 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 Serial1, Serial2, Serial3, Serial0, and I2cLcd.

```
9.14.3.21 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 Serial1, Serial2, Serial3, Serial0, and I2cLcd.

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

- Writer.h
- · Writer.cpp

166 Class Documentation

# **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.

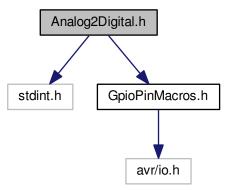
If when building your project you get link-time errors about undefined references to symbols of the form  $\__cxa\_XXX$  (e.g.,  $\__cxa\_pure\_virtual$ ), then you should link your project against abi.cpp (there is no need to include abi.h in any of your sources.

If you don't encounter such errors, you can completely disregard both abi.h and abi.cpp.

# 10.2 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**

# **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.

### 10.2.1 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.

### 10.2.2 Macro Definition Documentation

10.2.2.1 #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().

# 10.2.3 Enumeration Type Documentation

### 10.2.3.1 enum A2DVoltageReference

Constants representing voltage references.

### **Enumerator**

**kA2dReferenceAREF** Reference is AREF pin, internal VREF turned off.

kA2dReferenceAVCC Reference is AVCC pin, internal VREF turned off.

kA2dReference11V Reference is internal 1.1V VREF.

kA2dReference256V Reference is internal 2.56V VREF (only available on ATmega2560)

### 10.2.4 Function Documentation

10.2.4.1 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.

10.2.4.2 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().

10.2.4.3 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 makeGpioVarFromGpioPin←
Analog()).

### Returns

an value between 0 and 1023.

### Note

Before calling this function must fist initialize the analog-to-digital sub-system by calling initA2D().

10.2.4.4 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.

```
10.2.4.5 void setA2DVoltageReference11V() [inline]
```

Set the voltage reference for the analog-to-digital system to AREF.

This is an inline synonym for setA2DVoltageReference( kA2dReference11V )

```
10.2.4.6 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).

```
10.2.4.7 void setA2DVoltageReferenceAREF( ) [inline]
```

Set the voltage reference for the analog-to-digital system to AREF.

This is an inline synonym for setA2DVoltageReference( kA2dReferenceAREF )

```
10.2.4.8 void setA2DVoltageReferenceAVCC() [inline]
```

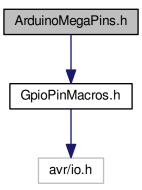
Set the voltage reference for the analog-to-digital system to AREF.

This is an inline synonym for setA2DVoltageReference( kA2dReferenceAVCC )

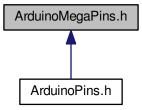
# 10.3 ArduinoMegaPins.h File Reference

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:



# 10.3.1 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.

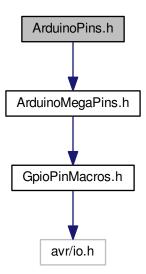
The standard Arduino Uno analog pins will be defined as pPinA00 through pPinA15.

Additionally, the I2C SDA and SCL pins are also defined as pSDA and pSCL (these are synonyms for pPin20 and pPin21, respectively).

# 10.4 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:



### 10.4.1 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).

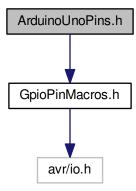
The standard Arduino digital pins will be defined in the form pPinNN (where NN = 00 through 13 for Arduino Uno, and 00 through 53 for Arduino Mega).

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).

# 10.5 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:



# 10.5.1 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 pPin13.

The standard Arduino Uno analog pins will be defined as pPinA00 through pPinA07.

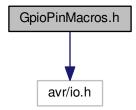
Additionally, the I2C SDA and SCL pins are also defined as pSDA and pSCL (these are synonyms for pPinA04 and pPinA05, respectively).

# 10.6 GpioPinMacros.h File Reference

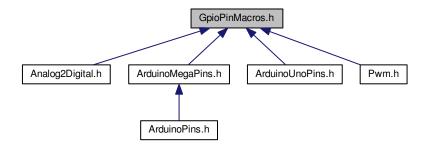
This file contains the primary macros for naming and manipulating GPIO pin names.

#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 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 correspoinding 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 correspoinding 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**

### **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 correspoinding the PINn bit).

void writeGpioPinDigitalV (const GpioPinVariable &pinVar, bool value)

Write a value the GPIO pin (i.e., set or clear the correspoinding 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).

### 10.6.1 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.

### 10.6.2 Macro Definition Documentation

10.6.2.1 #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).

```
10.6.2.2 #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)
```

```
10.6.2.3 #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)
```

```
10.6.2.4 #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

```
10.6.2.5 #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)

```
10.6.2.6 #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

```
10.6.2.7 #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)
```

```
10.6.2.8 #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)
```

```
10.6.2.9 #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.

10.6.2.10 #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)

```
10.6.2.11 #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.

```
10.6.2.12 #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().

```
10.6.2.13 #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().

10.6.2.14 #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.

10.6.2.15 #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.

10.6.2.16 #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.

10.6.2.17 #define readGpioPinDigital( pinName )

Read the value of the GPIO pin (i.e., return the value of correspoinding 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

10.6.2.18 #define setGpioPinHigh( pinName )

Write a 1 to the GPIO pin (i.e., set the correspoinding the PORTn bit).

• pinName a GPIO pin name macro generated by either GpioPin(), GpioPinAnalog(), or GpioPinPwm().

10.6.2.19 #define setGpioPinLow( pinName )

Write a 0 the GPIO pin (i.e., clear the correspoinding the PORTn bit).

• pinName a GPIO pin name macro generated by either GpioPin(), GpioPinAnalog(), or GpioPinPwm().

10.6.2.20 #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().

10.6.2.21 #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().

10.6.2.22 #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().

10.6.2.23 #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.

# 10.6.3 Enumeration Type Documentation

10.6.3.1 anonymous enum

Constants for digital values representing LOW and HIGH.

**Enumerator** 

kDigitalLow Value representing digital LOW.kDigitalHigh Value representing digital HIGH.

### 10.6.4 Function Documentation

10.6.4.1 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.

10.6.4.2 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.

10.6.4.3 bool readGpioPinDigitalV (const GpioPinVariable & pinVar) [inline]

Read the value of the GPIO pin (i.e., return the value of correspoinding the PINn bit).

• pinVar a GPIO pin variable of type GpioPinVariable.

### Returns

0 (false) or 1 (true)

10.6.4.4 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.

10.6.4.5 void setGpioPinLowV (const GpioPinVariable & pinVar) [inline]

Write a 0 to the GPIO pin (i.e., clear the correspoinding the PORTn bit).

• pinVar aa GPIO pin variable of type GpioPinVariable.

```
10.6.4.6 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.

```
10.6.4.7 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.

```
10.6.4.8 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.

```
10.6.4.9 void writeGpioPinDigitalV (const GpioPinVariable & pinVar, bool value) [inline]
```

Write a value the GPIO pin (i.e., set or clear the correspoinding the PORTn bit).

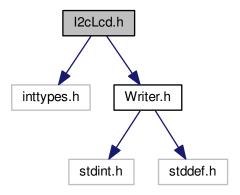
- pinVar a GPIO pin variable of type GpioPinVariable.
- val the value to be written: 0 means to clear the GPIO pin; any other value means to set it.

# 10.7 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 assocaited with such devices.

# 10.7.1 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.

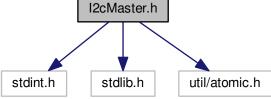
To use these features, include I2cLcd.h in your source code and link against I2cLcd.cpp and I2cMaster.cpp.

# 10.8 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:
```

I2cMaster.h



# **Namespaces**

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**

### **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 l2cMaster::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 l2cMaster::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 l2cMaster::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 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.

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).

### 10.8.1 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\_T \( \times \) X\_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., \( \times \)DI2C\_MASTER\_MAX\_TX\_MSG\_LEN=32 \( \times \)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.

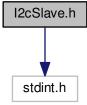
Note

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.

# 10.9 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**

• 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**

### **Functions**

uint8\_t l2cSlave::processl2cMessage (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=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 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).

### 10.9.1 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 recieving 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 l2cSlave.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.

# 10.10 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.

# 10.10.1 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.

### 10.10.2 Function Documentation

```
10.10.2.1 void initSystem ( )
```

This function initializes the microcontroller by clearing any bootloader settings, clearing all timers, and turning on interrupts.

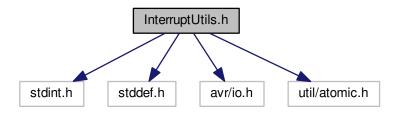
This function is generally called at the very beginning of main ().

# 10.11 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**

· Interrupts

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

#### **Enumerations**

## 10.11.1 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.

## 10.12 MemUtils.h File Reference

This file provides functions that provide information on the available memory in SRAM.

### **Namespaces**

MemUtils

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

#### **Functions**

• unsigned int MemUtils::freeRam ()

Get the number of free bytes remaining in SRAM.

unsigned int MemUtils::freeRamQuickEstimate ()

Get a quick estimate of the number of free bytes remaining in SRAM.

# 10.12.1 Detailed Description

This file provides functions that provide information on the available memory in SRAM.

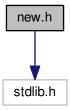
To use these functions, include MemUtils.h in your source code and link against MemUtils.cpp.

These functions are wrapped in namespace MemUtils to avoid namespace collisions.

## 10.13 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:



## 10.13.1 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

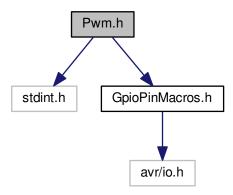
The AVRTools library does not itself make any use of heap storage or the new or delete operators.

10.14 Pwm.h File Reference 193

# 10.14 Pwm.h File Reference

This file provides functions that access the PWM capability of the ATmega328 and ATmega2560 microcontrollers.

```
#include <stdint.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.

# 10.14.1 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 init← PwmTimerN() 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

10.14 Pwm.h File Reference 195

Arduino Mega pin	ATmega2560 pin	Timer
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().

#### 10.14.2 Macro Definition Documentation

10.14.2.1 #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.

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().

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.

## 10.14.3 Function Documentation

10.14.3.1 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 ).

```
10.14.3.2 void clearTimer1 ( )
Clear timer1.
This function clears timer1, turning off the PWM functionality.
10.14.3.3 void clearTimer2 ( )
Clear timer2.
This function clears timer2, turning off the PWM functionality.
10.14.3.4 void clearTimer3 ( )
Clear timer3.
This function clears timer3, turning off the PWM functionality.
Note
     This function is only available on Arduino Mega (ATmega2560).
10.14.3.5 void clearTimer4 ( )
Clear timer4.
This function clears timer4, turning off the PWM functionality.
Note
     This function is only available on Arduino Mega (ATmega2560).
10.14.3.6 void clearTimer5 ( )
Clear timer5.
This function clears timer5, turning off the PWM functionality.
Note
     This function is only available on Arduino Mega (ATmega2560).
```

10.14 Pwm.h File Reference 197

```
10.14.3.7 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).

```
10.14.3.8 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)

```
10.14.3.9 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)

```
10.14.3.10 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).

```
10.14.3.11 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).

10.14 Pwm.h File Reference 199

10.14.3.12 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).

10.14.3.13 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.

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().

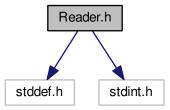
You can temporarily turn off PWM by writing a 0 to the pin with writePinPwm(pin, 0). In particular, this is how to turn off PWM to pins associated with timer0 when timer0 is also being used by the system clock.

# 10.15 Reader.h File Reference

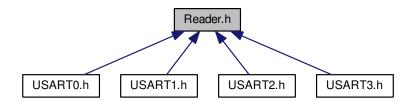
This file provides a generic interface to incoming data streams of any kind. It is designed around how serial streams are generally used, but can be used with any system that provides a sequential input of bytes that can be interpreted as strings and/or numbers.

```
#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).

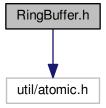
# 10.15.1 Detailed Description

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.

# 10.16 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 storgage of bytes, providing a single code base that is shared by all instances of this class.

## 10.16.1 Detailed Description

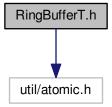
This file provides an efficient ring buffer implementation for storing bytes.

Ring buffers are particularly useful for memory constrained microcontrollers such as the ATmega328 and ATmega2650.

# 10.17 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.

# 10.17.1 Detailed Description

This file provides a very flexible, template-based ring buffer implementation.

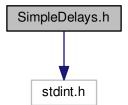
Ring buffers are versatile storage structures. This file provides a template-based ring buffer implementation that can store different kinds of objects in buffers of whatever size is needed.

# 10.18 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.

#### 10.18.1 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 or a 16 MHz clock cycle.

#### 10.18.2 Function Documentation

10.18.2.1 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 ( $\sim$ 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 ( $\sim$ 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 ( $\sim$ 1.5 microseconds). At 8 MHz delays of less than 12 quarter microseconds produce a delay of just under 12 quarter microseconds ( $\sim$ 3 microseconds).

The maximum delay is 65535 guarter microseconds (equal to 16,383.75 microseconds, or about 16.4 milliseconds).

nbr0fQuarters0fMicroSeconds 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 or 16 MHz.

For precision, this function is implemented directly in assembler.

10.18.2.2 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 or 16 MHz.

For precision, this function is implemented directly in assembler.

10.18.2.3 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.

• nbr0fMilliSeconds 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 or 16 MHz.

For precision, this function is implemented directly in assembler.

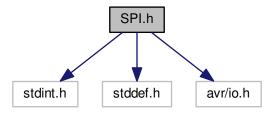
# 10.19 SPI.h File Reference

This file provides an interface to SPI subsystem available on the AVR ATMega328p (Arduino Uno) and ATMega2560 (Arduino Mega) microcontrollers.

10.19 SPI.h File Reference 205

```
#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**

• SPI

This namespace bundles an interface to the SPI hardware subsystem on the AVR ATMega328p (Arduino Uno) and AT← Mega2560 (Arduino Mega) microcontrollers. It provides logical cohesion for functions implement the Master portion of the SPI protocol and prevents namespace collisions.

#### **Enumerations**

#### **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.

## 10.19.1 Detailed Description

This file provides an interface to SPI subsystem available on the AVR ATMega328p (Arduino Uno) and ATMega2560 (Arduino Mega) microcontrollers.

# 10.20 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 elasped microseconds since the system clock was turned on.

· unsigned long millis ()

Return the number of elasped milliseconds since the system clock was turned on.

### 10.20.1 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.

### 10.20.2 Function Documentation

10.20.2.1 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.

10.20.2.2 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 or 16 MHz.

10.20.2.3 void delayMilliseconds (unsigned long ms)

Delay a certain number of milliseconds.

• m the number of milliseconds to delay.

```
10.20.2.4 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.

```
10.20.2.5 unsigned long micros ( )
```

Return the number of elasped 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.

```
10.20.2.6 unsigned long millis ( )
```

Return the number of elasped milliseconds since the system clock was turned on.

#### Returns

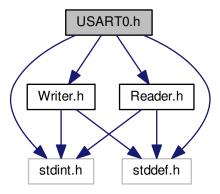
the number of elapsed milliseconds.

# 10.21 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 USARTO.

## **Namespaces**

• USART0

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

#### **Enumerations**

#### **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 USARTO.

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..

## 10.21.1 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 <code>USARTO\_RX\_ \leftrightarrow BUFFER\_SIZE</code> (for the receive buffer) and <code>USARTO\_TX\_BUFFER\_SIZE</code> (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file <code>USARTO.cpp</code>.

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.

## 10.21.2 Enumeration Type Documentation

#### 10.21.2.1 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_501 5 data bits, odd parity, 1 stop bit
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
```

kSerial\_802 8 data bits, odd parity, 2 stop bits 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\_501 5 data bits, odd parity, 1 stop bit **kSerial\_601** 6 data bits, odd parity, 1 stop bit kSerial\_701 7 data bits, odd parity, 1 stop bit **kSerial\_801** 8 data bits, odd parity, 1 stop bit kSerial\_502 5 data bits, odd parity, 2 stop bits kSerial\_602 6 data bits, odd parity, 2 stop bits kSerial\_702 7 data bits, odd parity, 2 stop bits kSerial\_802 8 data bits, odd parity, 2 stop bits 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\_501 5 data bits, odd parity, 1 stop bit

```
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802 8 data bits, odd parity, 2 stop bits
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_501 5 data bits, odd parity, 1 stop bit
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802 8 data bits, odd parity, 2 stop bits
```

## 10.22 USARTOMinimal.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 USARTO.

• unsigned char receiveUSART0 ()

Receive a byte on USARTO.

• void releaseUSART0 ()

Release USART0, making pins 0 and 1 again available for non-USART use.

## 10.22.1 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 sigle 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.

### 10.22.2 Function Documentation

10.22.2.1 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 (A← Tmega2560 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).

```
10.22.2.2 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.
10.22.2.3 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().
10.22.2.4 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.
10.22.2.5 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.
```

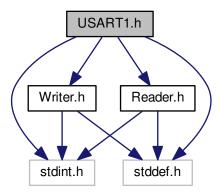
data the null-terminated string to be transmitted.

# 10.23 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**

• USART1

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

## **Enumerations**

# **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..

### 10.23.1 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 <code>USART1\_RX\_ \leftrightarrow BUFFER\_SIZE</code> (for the receive buffer) and <code>USART1\_TX\_BUFFER\_SIZE</code> (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file <code>USART1.cpp</code>.

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 USAR ← T1\_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.

## 10.23.2 Enumeration Type Documentation

#### 10.23.2.1 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_501 5 data bits, odd parity, 1 stop bit
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802 8 data bits, odd parity, 2 stop bits
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\_501** 5 data bits, odd parity, 1 stop bit kSerial\_601 6 data bits, odd parity, 1 stop bit kSerial\_701 7 data bits, odd parity, 1 stop bit kSerial\_801 8 data bits, odd parity, 1 stop bit kSerial\_502 5 data bits, odd parity, 2 stop bits kSerial\_602 6 data bits, odd parity, 2 stop bits kSerial\_702 7 data bits, odd parity, 2 stop bits kSerial\_802 8 data bits, odd parity, 2 stop bits 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\_501 5 data bits, odd parity, 1 stop bit kSerial\_601 6 data bits, odd parity, 1 stop bit kSerial\_701 7 data bits, odd parity, 1 stop bit kSerial\_801 8 data bits, odd parity, 1 stop bit kSerial\_502 5 data bits, odd parity, 2 stop bits kSerial\_602 6 data bits, odd parity, 2 stop bits kSerial\_702 7 data bits, odd parity, 2 stop bits

```
kSerial_802 8 data bits, odd parity, 2 stop bits
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_501 5 data bits, odd parity, 1 stop bit
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802 8 data bits, odd parity, 2 stop bits
```

# 10.24 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.

## 10.24.1 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 sigle 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.

#### 10.24.2 Function Documentation

10.24.2.1 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).

10.24.2.2 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).

```
10.24.2.3 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).
10.24.2.4 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).
10.24.2.5 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

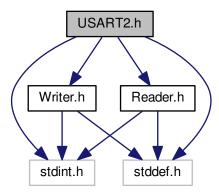
This function is only available on Arduino Mega (ATmega2560).

# 10.25 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**

• USART2

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

## **Enumerations**

# **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..

### 10.25.1 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 <code>USART2\_RX\_ \leftrightarrow BUFFER\_SIZE</code> (for the receive buffer) and <code>USART2\_TX\_BUFFER\_SIZE</code> (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file <code>USART2.cpp</code>.

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 USAR ← T2\_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.

## 10.25.2 Enumeration Type Documentation

#### 10.25.2.1 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_501 5 data bits, odd parity, 1 stop bit
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802 8 data bits, odd parity, 2 stop bits
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\_501** 5 data bits, odd parity, 1 stop bit **kSerial\_601** 6 data bits, odd parity, 1 stop bit kSerial\_701 7 data bits, odd parity, 1 stop bit kSerial\_801 8 data bits, odd parity, 1 stop bit kSerial\_502 5 data bits, odd parity, 2 stop bits kSerial\_602 6 data bits, odd parity, 2 stop bits kSerial\_702 7 data bits, odd parity, 2 stop bits kSerial\_802 8 data bits, odd parity, 2 stop bits 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\_501 5 data bits, odd parity, 1 stop bit **kSerial\_601** 6 data bits, odd parity, 1 stop bit kSerial\_701 7 data bits, odd parity, 1 stop bit kSerial\_801 8 data bits, odd parity, 1 stop bit kSerial\_502 5 data bits, odd parity, 2 stop bits kSerial\_602 6 data bits, odd parity, 2 stop bits kSerial\_702 7 data bits, odd parity, 2 stop bits

```
kSerial_802 8 data bits, odd parity, 2 stop bits
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_501 5 data bits, odd parity, 1 stop bit
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802 8 data bits, odd parity, 2 stop bits
```

# 10.26 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.

# 10.26.1 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 sigle 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.

#### 10.26.2 Function Documentation

10.26.2.1 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).

10.26.2.2 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).

```
10.26.2.3 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).
10.26.2.4 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).
10.26.2.5 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
```

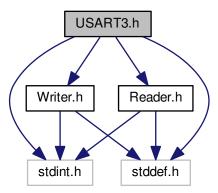
This function is only available on Arduino Mega (ATmega2560).

# 10.27 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**

• USART3

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

#### **Enumerations**

# **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..

# 10.27.1 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 <code>USART3\_RX\_ \leftrightarrow BUFFER\_SIZE</code> (for the receive buffer) and <code>USART3\_TX\_BUFFER\_SIZE</code> (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file <code>USART3.cpp</code>.

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 USAR ← T3\_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.

# 10.27.2 Enumeration Type Documentation

#### 10.27.2.1 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_501 5 data bits, odd parity, 1 stop bit
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802 8 data bits, odd parity, 2 stop bits
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\_501** 5 data bits, odd parity, 1 stop bit **kSerial\_601** 6 data bits, odd parity, 1 stop bit **kSerial\_701** 7 data bits, odd parity, 1 stop bit kSerial\_801 8 data bits, odd parity, 1 stop bit kSerial\_502 5 data bits, odd parity, 2 stop bits kSerial\_602 6 data bits, odd parity, 2 stop bits kSerial\_702 7 data bits, odd parity, 2 stop bits kSerial\_802 8 data bits, odd parity, 2 stop bits 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\_501 5 data bits, odd parity, 1 stop bit **kSerial\_601** 6 data bits, odd parity, 1 stop bit kSerial\_701 7 data bits, odd parity, 1 stop bit kSerial\_801 8 data bits, odd parity, 1 stop bit kSerial\_502 5 data bits, odd parity, 2 stop bits kSerial\_602 6 data bits, odd parity, 2 stop bits kSerial\_702 7 data bits, odd parity, 2 stop bits

```
kSerial_802 8 data bits, odd parity, 2 stop bits
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_501 5 data bits, odd parity, 1 stop bit
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802 8 data bits, odd parity, 2 stop bits
```

# 10.28 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.

# 10.28.1 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 sigle 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.

#### 10.28.2 Function Documentation

10.28.2.1 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).

10.28.2.2 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).

```
10.28.2.3 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).
10.28.2.4 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).
10.28.2.5 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.
```

Note

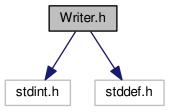
This function is only available on Arduino Mega (ATmega2560).

• data the null-terminated string to be transmitted.

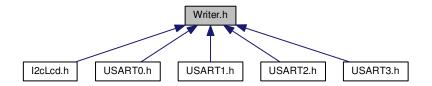
# 10.29 Writer.h File Reference

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).

# 10.29.1 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.

# Index

10DV-ltD-f	ala a eTira a eA
A2DVoltageReference	clearTimer4
Analog2Digital.h, 169	Pwm.h, 196
abi.h, 167	clearTimer5
Analog2Digital.h, 168	Pwm.h, 196
A2DVoltageReference, 169	command
initA2D, 170	I2cLcd, 71
kA2dReference11V, 169	configure
kA2dReference256V, 169	SPI, 48
kA2dReferenceAREF, 169	
kA2dReferenceAVCC, 169	delay
readA2D, 170	SystemClock.h, 206
readGpioPinAnalog, 169	delayMicroseconds
readGpioPinAnalogV, 170	SystemClock.h, 206
setA2DVoltageReference, 170	delayMilliseconds
	SystemClock.h, 207
setA2DVoltageReference11V, 171	-
setA2DVoltageReference256V, 171	delayQuartersOfMicroSeconds
setA2DVoltageReferenceAREF, 171	SimpleDelays.h, 203
setA2DVoltageReferenceAVCC, 171	delayTenthsOfSeconds
ArduinoMegaPins.h, 172	SimpleDelays.h, 203
ArduinoPins.h, 173	delayWholeMilliSeconds
ArduinoUnoPins.h, 174	SimpleDelays.h, 204
available	disable
Reader, 82	SPI, 48
Serial0, 96	discardFromFront
Serial1, 112	RingBufferT, 91
Serial2, 128	displayBottomRow
Serial3, 143	I2cLcd, 71
USART0, 51	displayTopRow
USART1, 54	I2cLcd, 71
USART2, 58	,
USART3, 61	enable
00/1110, 01	SPI, 48
busy	ExternalInterrupts
I2cMaster, 34	Interrupts, 44
I2cSlave, 41	ExternalOff
ByteOrder	Interrupts::ExternalOff, 66
•	interruptsExternation, 00
SPI, 47	find
clearTimer0	Reader, 82
Pwm.h, 195	Serial0, 96, 97
clearTimer1	
	Serial1, 112
Pwm.h, 195	Serial2, 128
clearTimer2	Serial3, 143
Pwm.h, 196	findUntil
clearTimer3	Reader, 83
Pwm.h, 196	Serial0, 97

Serial1, 113	kDigitalLow, 182
Serial2, 128	makeGpioVarFromGpioPin, 180
Serial3, 144	makeGpioVarFromGpioPinAnalog, 180
flush	makeGpioVarFromGpioPinPwm, 181
USARTO, 51	readGpioPinDigital, 181
USART1, 54	readGpioPinDigitalV, 183
USART2, 58	setGpioPinHigh, 181
USART3, 61	setGpioPinHighV, 183
freeRam	setGpioPinLow, 181
MemUtils, 45	setGpioPinLowV, 183
freeRamQuickEstimate	setGpioPinModeInput, 182
MemUtils, 45	setGpioPinModeInputPullup, 182
	setGpioPinModeInputPullupV, 183
getGpioADC	setGpioPinModeInputV, 184
GpioPinMacros.h, 177	setGpioPinModeOutput, 182
getGpioCOM	setGpioPinModeOutputV, 184
GpioPinMacros.h, 177	writeGpioPinDigital, 182
getGpioDDR	writeGpioPinDigitalV, 184
GpioPinMacros.h, 178	GpioPinPwm
getGpioMASK	GpioPinMacros.h, 180
GpioPinMacros.h, 178	GpioPinVariable, 66
getGpioOCR	apier invaliable, ee
GpioPinMacros.h, 178	I2cBusSpeed
getGpioPIN	I2cMaster, 33
GpioPinMacros.h, 178	I2cSlave, 41
getGpioPORT	12colave, 41
GpioPinMacros.h, 179	command, 71
getGpioTCCR	displayBottomRow, 71
- ,	
GpioPinMacros.h, 179	displayTopRow, 71
getSpcr	init, 72
SPI::SPISettings, 156	IntegerOutputBase, 71
getSpsr SPIUSPISattings 156	kBacklight_Blue, 71
SPI::SPISettings, 156	kBacklight_Green, 71
GpioPin	kBacklight_Red, 71
GpioPinMacros.h, 179	kBacklight_Teal, 71
GpioPinAnalog	kBacklight_Violet, 71
GpioPinMacros.h, 179	kBacklight_White, 71
GpioPinMacros.h, 174	kBacklight_Yellow, 71
getGpioADC, 177	kBin, 71
getGpioCOM, 177	kButton_Down, 70
getGpioDDR, 178	kButton_Left, 70
getGpioMASK, 178	kButton_Right, 70
getGpioOCR, 178	kButton_Select, 70
getGpioPIN, 178	kButton_Up, 70
getGpioPORT, 179	kDec, 71
getGpioTCCR, 179	kHex, 71
GpioPin, 179	kOct, 71
GpioPinAnalog, 179	print, 72-74
GpioPinPwm, 180	println, 74–77
isGpioPinModeInput, 180	readButtons, 77
isGpioPinModeInputV, 183	setBacklight, 77
isGpioPinModeOutput, 180	setCursor, 77
isGpioPinModeOutputV, 183	write, 78
kDigitalHigh, 182	I2cLcd.h, 184

I2cMaster, 31	I2cSlave, 41
busy, 34	init
I2cBusSpeed, 33	I2cLcd, 72
I2cPullups, 33	initA2D
I2cSendErrorCodes, 33	Analog2Digital.h, 170
I2cStatusCodes, 33	initPwmTimer0
kl2cBusFast, 33	Pwm.h, 196
kl2cBusSlow, 33	initPwmTimer1
kl2cCompletedOk, 34	Pwm.h, 197
kl2cErrMsgTooLong, 33	initPwmTimer2
kl2cErrNullStatusPtr, 33	Pwm.h, 197
kl2cErrReadWithoutStorage, 33	initPwmTimer3
kl2cErrTxBufferFull, 33	Pwm.h, 198
kl2cErrWriteWithoutData, 33	initPwmTimer4
kl2cError, 34	Pwm.h, 198
kl2cInProgress, 34	initPwmTimer5
kl2cNoError, 33	Pwm.h, 198
kl2cNotStarted, 34	initSystem
kPullupsOff, 33	InitSystem.h, 190
kPullupsOn, 33	InitSystem.h, 190
pullups, 34	initSystem, 190
readAsync, 34, 35	initSystemClock
readSync, 35	SystemClock.h, 207
start, 36	initUSART0
stop, 36	USART0Minimal.h, 213
writeAsync, 36, 37	initUSART1
writeSync, 38, 39	USART1Minimal.h, 220
I2cMaster.h, 185	initUSART2
I2cPullups	USART2Minimal.h, 227
I2cMaster, 33	initUSART3
I2cSlave, 41	USART3Minimal.h, 234
I2cSendErrorCodes	IntegerOutputBase
I2cMaster, 33	I2cLcd, 71
I2cSlave, 40	Serial0, 96
busy, 41	Serial1, 112
I2cBusSpeed, 41	Serial2, 127
I2cPullups, 41	Serial3, 143
I2cStatusCodes, 41	Writer, 159
kl2cBusFast, 41	InterruptUtils.h, 190
kl2cBusSlow, 41	Interrupts, 43
kl2cCompletedOk, 41	ExternalInterrupts, 44
kl2cError, 41	kExternalInterrupt0, 44
kl2cInProgress, 41	kExternalInterrupt1, 44
kl2cRxOverflow, 41	kExternalInterrupt2, 44
kl2cTxPartial, 41	kExternalInterrupt3, 44
kPullupsOff, 41	kExternalInterrupt4, 44
kPullupsOn, 41	kExternalInterrupt5, 44
processI2cMessage, 41	kExternalInterrupt6, 44
pullups, 42	kExternalInterrupt7, 44
start, 42	kExternalInterruptAll, 44
stop, 43	kPinChangeInterrupt0, 44
I2cSlave.h, 188	kPinChangeInterrupt1, 44
I2cStatusCodes	kPinChangeInterrupt2, 44
I2cMaster, 33	kPinChangeInterruptAll, 44
	in monangonitorraph iii, 4

PinChangeInterrupts, 44	Serial3, 143
Interrupts::AllOff, 65	Writer, 159
Interrupts::ExternalOff, 65	kButton_Down
ExternalOff, 66	I2cLcd, 70
Interrupts::PinChangeOff, 79	kButton_Left
PinChangeOff, 80	I2cLcd, 70
isEmpty	kButton_Right
RingBuffer, 89	I2cLcd, 70
RingBufferT, 91	kButton_Select
isFull	I2cLcd, 70
RingBuffer, 89	kButton_Up
RingBufferT, 91	I2cLcd, 70
isGpioPinModeInput	kDec
GpioPinMacros.h, 180	I2cLcd, 71
isGpioPinModeInputV	Serial0, 96
GpioPinMacros.h, 183	Serial1, 112
isGpioPinModeOutput	Serial2, 127
GpioPinMacros.h, 180	Serial3, 143
isGpioPinModeOutputV	Writer, 159
GpioPinMacros.h, 183	kDigitalHigh
isNotEmpty	GpioPinMacros.h, 182
RingBuffer, 89	kDigitalLow
RingBufferT, 91	GpioPinMacros.h, 182
isNotFull	kExternalInterrupt0
RingBuffer, 89	Interrupts, 44
RingBufferT, 92	kExternalInterrupt1
	Interrupts, 44
kA2dReference11V	kExternalInterrupt2
Analog2Digital.h, 169	Interrupts, 44
kA2dReference256V	kExternalInterrupt3
Analog2Digital.h, 169	Interrupts, 44
kA2dReferenceAREF	kExternalInterrupt4
Analog2Digital.h, 169	Interrupts, 44
kA2dReferenceAVCC	kExternalInterrupt5
Analog2Digital.h, 169	Interrupts, 44
kBacklight_Blue	kExternalInterrupt6
I2cLcd, 71	Interrupts, 44
kBacklight_Green	kExternalInterrupt7
I2cLcd, 71	Interrupts, 44
kBacklight_Red	kExternalInterruptAll
I2cLcd, 71	Interrupts, 44
kBacklight_Teal	kHex
I2cLcd, 71	I2cLcd, 71
kBacklight_Violet	Serial0, 96
I2cLcd, 71	Serial1, 112
kBacklight_White	Serial2, 127
I2cLcd, 71	Serial3, 143
kBacklight_Yellow	Writer, 159
I2cLcd, 71	kl2cBusFast
kBin	I2cMaster, 33
I2cLcd, 71	I2cSlave, 41
Serial0, 96	kl2cBusSlow
Serial 1, 112	I2cMaster, 33
Serial2, 127	I2cSlave, 41
JOHAIE, IEI	12001ave, +1

kl2cCompletedOk	USART1.h, 217-219
I2cMaster, 34	USART2.h, 224-226
I2cSlave, 41	USART3.h, 231-233
kl2cErrMsgTooLong	kSerial_5E2
I2cMaster, 33	USART0.h, 210-212
kl2cErrNullStatusPtr	USART1.h, 217-219
I2cMaster, 33	USART2.h, 224-226
kl2cErrReadWithoutStorage	USART3.h, 231-233
I2cMaster, 33	kSerial_5N1
kl2cErrTxBufferFull	USART0.h, 210-212
I2cMaster, 33	USART1.h, 217-219
kl2cErrWriteWithoutData	USART2.h, 224-226
I2cMaster, 33	USART3.h, 231–233
kl2cError	kSerial_5N2
I2cMaster, 34	USART0.h, 210-212
I2cSlave, 41	USART1.h, 217-219
kl2cInProgress	USART2.h, 224–226
I2cMaster, 34	USART3.h, 231–233
I2cSlave, 41	kSerial_5O1
kl2cNoError	USART0.h, 210-212
I2cMaster, 33	USART1.h, 217-219
kl2cNotStarted	USART2.h, 224-226
I2cMaster, 34	USART3.h, 231–233
kl2cRxOverflow	kSerial_5O2
I2cSlave, 41	USART0.h, 210-212
kl2cTxPartial	USART1.h, 217–219
I2cSlave, 41	USART2.h, 224–226
kLsbFirst	USART3.h, 231–233
SPI, 47	kSerial_6E1
kMsbFirst	USARTO.h, 210–212
SPI, 47	USART1.h, 217–219
kOct	USART2.h, 224–226
I2cLcd, 71	USART3.h, 231–233
Serialo, 96	kSerial_6E2
Serial1, 112	USART0.h, 210–212
Serial2, 127	USART1.h, 217–219
Serial3, 143	USART2.h, 224–226
Writer, 159	USART3.h, 231–233
kPinChangeInterrupt0	kSerial_6N1
Interrupts, 44	USART0.h, 210–212
kPinChangeInterrupt1	USART1.h, 217–219
Interrupts, 44	USART2.h, 224–226
kPinChangeInterrupt2	USART3.h, 231–233
Interrupts, 44	kSerial_6N2
kPinChangeInterruptAll	USART0.h, 210–212
Interrupts, 44	USART1.h, 217–219
kPullupsOff	USART2.h, 224–226
I2cMaster, 33	USART3.h, 231–233
I2cSlave, 41	kSerial_6O1
kPullupsOn	USART0.h, 210–212
I2cMaster, 33	USART1.h, 217–219
I2cSlave, 41	USART2.h, 224–226
kSerial_5E1	USART3.h, 231–233
USART0.h, 210–212	kSerial_6O2

USART0.h, 210-212	kSerial_8O1
USART1.h, 217-219	USART0.h, 210-212
USART2.h, 224-226	USART1.h, 217-219
USART3.h, 231-233	USART2.h, 224-226
kSerial_7E1	USART3.h, 231-233
USART0.h, 210-212	kSerial_8O2
USART1.h, 217-219	USART0.h, 210-212
USART2.h, 224-226	USART1.h, 217-219
USART3.h, 231–233	USART2.h, 224-226
kSerial 7E2	USART3.h, 231-233
USART0.h, 210–212	kSpiMode0
USART1.h, 217–219	SPI, 47
USART2.h, 224–226	kSpiMode1
USART3.h, 231–233	SPI, 47
	kSpiMode2
kSerial_7N1	SPI, 47
USART0.h, 210–212	kSpiMode3
USART1.h, 217–219	•
USART2.h, 224–226	SPI, 47
USART3.h, 231–233	makeGpioVarFromGpioPin
kSerial_7N2	GpioPinMacros.h, 180
USART0.h, 210-212	makeGpioVarFromGpioPinAnalog
USART1.h, 217-219	
USART2.h, 224-226	GpioPinMacros.h, 180
USART3.h, 231-233	makeGpioVarFromGpioPinPwm
kSerial_701	GpioPinMacros.h, 181
USART0.h, 210-212	MemUtils, 44
USART1.h, 217-219	freeRam, 45
USART2.h, 224-226	freeRamQuickEstimate, 45
USART3.h, 231–233	MemUtils.h, 191
kSerial 702	micros
USART0.h, 210–212	SystemClock.h, 207
USART1.h, 217–219	millis
USART2.h, 224–226	SystemClock.h, 207
USART3.h, 231–233	
,	new.h, 192
kSerial_8E1	
USART0.h, 210–212	peek
USART1.h, 217–219	Reader, 83
USART2.h, 224–226	RingBuffer, 89
USART3.h, 231–233	RingBufferT, 92
kSerial_8E2	Serial0, 98
USART0.h, 210-212	Serial1, 113
USART1.h, 217–219	Serial2, 129
USART2.h, 224-226	Serial3, 144
USART3.h, 231-233	USART0, 51
kSerial_8N1	USART1, 54
USART0.h, 210-212	USART2, 58
USART1.h, 217-219	USART3, 61
USART2.h, 224-226	PinChangeInterrupts
USART3.h, 231–233	Interrupts, 44
kSerial 8N2	PinChangeOff
USART0.h, 210–212	Interrupts::PinChangeOff, 80
USART1.h, 217–219	print
USART2.h, 224–226	I2cLcd, 72–74
USART3.h, 231–233	Serial0, 98–100
00/1110.11, <u>201</u> -200	Ochlaio, Jo-100

Serial1, 114–116	readBytes
Serial2, 129–131	Reader, 84
Serial3, 145–147	Serial0, 104
Writer, 159–161	Serial1, 119
println	Serial2, 135
I2cLcd, 74-77	Serial3, 150
Serial0, 101–103	readBytesUntil
Serial1, 116–118	Reader, 84, 85
Serial2, 132–134	Serial0, 104
Serial3, 147–149	Serial1, 119, 120
Writer, 161–164	Serial2, 135
processl2cMessage	Serial3, 150, 151
I2cSlave, 41	readFloat
pull	Reader, 85
RingBuffer, 89	Serial0, 105
RingBufferT, 92	Serial1, 120
pullups	Serial2, 136
I2cMaster, 34	Serial3, 151
I2cSlave, 42	readGpioPinAnalog
push	Analog2Digital.h, 169
RingBuffer, 90	readGpioPinAnalogV
RingBufferT, 92	Analog2Digital.h, 170
Pwm.h, 193	readGpioPinDigital
clearTimer0, 195	GpioPinMacros.h, 181
clearTimer1, 195	readGpioPinDigitalV
clearTimer2, 196	GpioPinMacros.h, 183
clearTimer3, 196	readLine
clearTimer4, 196	Reader, 86
clearTimer5, 196	Serial0, 105
initPwmTimer0, 196	Serial1, 121
initPwmTimer1, 197	Serial2, 136
initPwmTimer2, 197	Serial3, 152
initPwmTimer3, 198	readLong
initPwmTimer4, 198	Reader, 86
initPwmTimer5, 198	Serial0, 106
writeGpioPinPwm, 195	Serial1, 121
writeGpioPinPwmV, 199	Serial2, 137
,	Serial3, 152
read	readSync
Reader, 84	I2cMaster, 35
Serial0, 103	Reader, 80
Serial1, 119	available, 82
Serial2, 134	find, 82
Serial3, 150	findUntil, 83
USARTO, 51	peek, 83
USART1, 55	read, 84
USART2, 58	readBytes, 84
USART3, 62	readBytesUntil, 84, 85
readA2D	readFloat, 85
Analog2Digital.h, 170	readLine, 86
readAsync	readLong, 86
12cMaster, 34, 35	setTimeout, 87
readButtons	Reader.h, 200
I2cLcd, 77	receiveUSART0
, , ,	

USART0Minimal.h, 213	kSpiMode3, 47
receiveUSART1	SpiMode, 47
USART1Minimal.h, 220	transmit, 49
receiveUSART2	transmit16, 49
USART2Minimal.h, 227	transmit32, 49
receiveUSART3	Serial0, 93
USART3Minimal.h, 234	available, 96
releaseUSART0	find, 96, 97
USART0Minimal.h, 214	findUntil, 97
releaseUSART1	IntegerOutputBase, 96
USART1Minimal.h, 220	kBin, 96
releaseUSART2	kDec, 96
USART2Minimal.h, 227	kHex, 96
releaseUSART3	kOct, 96
USART3Minimal.h, 234	peek, 98
RingBuffer, 87	print, 98–100
isEmpty, 89	println, 101–103
isFull, 89	read, 103
isNotEmpty, 89	readBytes, 104
isNotFull, 89	readBytesUntil, 104
peek, 89	readFloat, 105
pull, 89	readLine, 105
push, 90	readLong, 106
RingBuffer, 88	setTimeout, 106
RingBuffer.h, 201	start, 107
RingBufferT< T, N, SIZE >, 90	stop, 107
RingBufferT.h, 201	write, 107, 108
<del>-</del>	
RingBufferT	Serial1, 109
RingBufferT discardFromFront, 91	Serial1, 109 available, 112
RingBufferT discardFromFront, 91 isEmpty, 91	Serial1, 109 available, 112 find, 112
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91	Serial1, 109 available, 112 find, 112 findUntil, 113
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 push, 92	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 push, 92 SPI.h, 204	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings, 156	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings, 156 SPISettings, 156	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readFloat, 120
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings SPI::SPISettings, 156	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readFloat, 120 readLine, 121
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings SPISettings, 156 SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readFloat, 120 readLine, 121 readLong, 121
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45 ByteOrder, 47	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readLine, 121 readLong, 121 setTimeout, 122
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings, 156 SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45 ByteOrder, 47 configure, 48	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytes, 119 readBytesUntil, 119, 120 readLine, 121 readLong, 121 setTimeout, 122 start, 122
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings, 156 SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45 ByteOrder, 47 configure, 48 disable, 48	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytes, 119 readBytesUntil, 119, 120 readFloat, 120 readLine, 121 readLong, 121 setTimeout, 122 start, 122 stop, 122
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45 ByteOrder, 47 configure, 48 disable, 48 enable, 48	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readFloat, 120 readLine, 121 readLong, 121 setTimeout, 122 start, 122 stop, 122 write, 122, 123
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45 ByteOrder, 47 configure, 48 disable, 48 enable, 48 kLsbFirst, 47	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readFloat, 120 readLine, 121 readLong, 121 setTimeout, 122 start, 122 stop, 122 write, 122, 123 Serial2, 124
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings SPISettings, 156 SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45 ByteOrder, 47 configure, 48 disable, 48 enable, 48 enable, 48 kLsbFirst, 47 kMsbFirst, 47	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readFloat, 120 readLine, 121 readLong, 121 setTimeout, 122 start, 122 stop, 122 write, 122, 123 Serial2, 124 available, 128
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings, 156 SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45 ByteOrder, 47 configure, 48 disable, 48 enable, 48 enable, 48 kLsbFirst, 47 kMsbFirst, 47 kSpiMode0, 47	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readLine, 121 readLong, 121 setTimeout, 122 start, 122 stop, 122 write, 122, 123 Serial2, 124 available, 128 find, 128
RingBufferT discardFromFront, 91 isEmpty, 91 isFull, 91 isNotEmpty, 91 isNotFull, 92 peek, 92 pull, 92 push, 92  SPI.h, 204 SPI::SPISettings, 155 getSpcr, 156 getSpsr, 156 SPISettings SPISettings, 156 SPISettings SPI::SPISettings, 156 SPISettings SPI::SPISettings, 156 SPI, 45 ByteOrder, 47 configure, 48 disable, 48 enable, 48 enable, 48 kLsbFirst, 47 kMsbFirst, 47	Serial1, 109 available, 112 find, 112 findUntil, 113 IntegerOutputBase, 112 kBin, 112 kDec, 112 kHex, 112 kOct, 112 peek, 113 print, 114–116 println, 116–118 read, 119 readBytes, 119 readBytesUntil, 119, 120 readFloat, 120 readLine, 121 readLong, 121 setTimeout, 122 start, 122 stop, 122 write, 122, 123 Serial2, 124 available, 128

kBin, 127	GpioPinMacros.h, 181
kDec, 127	setGpioPinHighV
kHex, 127	GpioPinMacros.h, 183
kOct, 127	setGpioPinLow
peek, 129	GpioPinMacros.h, 181
print, 129-131	setGpioPinLowV
println, 132–134	GpioPinMacros.h, 183
read, 134	setGpioPinModeInput
readBytes, 135	GpioPinMacros.h, 182
readBytesUntil, 135	setGpioPinModeInputPullup
readFloat, 136	GpioPinMacros.h, 182
readLine, 136	setGpioPinModeInputPullupV
readLong, 137	GpioPinMacros.h, 183
setTimeout, 137	setGpioPinModeInputV
start, 138	GpioPinMacros.h, 184
stop, 138	setGpioPinModeOutput
write, 138, 139	GpioPinMacros.h, 182
Serial3, 140	setGpioPinModeOutputV
available, 143	GpioPinMacros.h, 184
find, 143	setTimeout
findUntil, 144	Reader, 87
IntegerOutputBase, 143	Serial0, 106
kBin, 143	Serial1, 122
kDec, 143	Serial2, 137
kHex, 143	Serial3, 153
kOct, 143	SimpleDelays.h, 202
peek, 144	•
·	delayQuartersOfMicroSeconds, 203
print, 145–147	delayTenthsOfSeconds, 203
println, 147–149	delayWholeMilliSeconds, 204
read, 150	SpiMode CPL 47
readBytes, 150	SPI, 47
readBytesUntil, 150, 151	start
readFloat, 151	I2cMaster, 36
readLine, 152	I2cSlave, 42
readLong, 152	Serial0, 107
setTimeout, 153	Serial1, 122
start, 153	Serial2, 138
stop, 153	Serial3, 153
write, 153, 154	USART0, 51
setA2DVoltageReference	USART1, 55
Analog2Digital.h, 170	USART2, 58
setA2DVoltageReference11V	USART3, 62
Analog2Digital.h, 171	stop
setA2DVoltageReference256V	I2cMaster, 36
Analog2Digital.h, 171	I2cSlave, 43
setA2DVoltageReferenceAREF	Serial0, 107
Analog2Digital.h, 171	Serial1, 122
setA2DVoltageReferenceAVCC	Serial2, 138
Analog2Digital.h, 171	Serial3, 153
setBacklight	USARTO, 51
I2cLcd, 77	USART1, 55
setCursor	USART2, 59
I2cLcd, 77	USART3, 62
setGpioPinHigh	SystemClock.h, 206
. 3	, ,

delay, 206	UsartSerialConfiguration, 210
delayMicroseconds, 206	USART0Minimal.h, 212
delayMilliseconds, 207	initUSART0, 213
initSystemClock, 207	receiveUSART0, 213
micros, 207	releaseUSART0, 214
millis, 207	transmitUSART0, 214
	USART1, 53
transmit	available, 54
SPI, 49	flush, 54
transmit16	peek, 54
SPI, 49	read, 55
transmit32	start, 55
SPI, 49	stop, 55
transmitUSART0	write, 55, 56
USART0Minimal.h, 214	USART1.h, 215
transmitUSART1	kSerial_5E1, 217–219
USART1Minimal.h, 221	kSerial_5E2, 217–219
transmitUSART2	kSerial_5N1, 217–219
USART2Minimal.h, 228	kSerial_5N2, 217-219
transmitUSART3	kSerial_501, 217–219
USART3Minimal.h, 235	kSerial_502, 217-219
	kSerial_6E1, 217–219
USARTO, 50	kSerial_6E2, 217–219
available, 51	kSerial 6N1, 217–219
flush, 51	kSerial 6N2, 217–219
peek, 51	kSerial_601, 217–219
read, 51	kSerial_602, 217–219
start, 51	kSerial_7E1, 217–219
stop, 51	kSerial 7E2, 217–219
write, 52, 53	<del>-</del> · · ·
USART0.h, 208	kSerial_7N1, 217–219
kSerial_5E1, 210–212	kSerial_7N2, 217–219
kSerial_5E2, 210-212	kSerial_701, 217–219
kSerial_5N1, 210-212	kSerial_702, 217–219
kSerial_5N2, 210-212	kSerial_8E1, 217–219
kSerial_5O1, 210-212	kSerial_8E2, 217–219
kSerial_502, 210-212	kSerial_8N1, 217–219
kSerial_6E1, 210-212	kSerial_8N2, 217–219
kSerial_6E2, 210-212	kSerial_801, 217–219
kSerial_6N1, 210-212	kSerial_802, 217–219
kSerial_6N2, 210-212	UsartSerialConfiguration, 217
kSerial_6O1, 210-212	USART1Minimal.h, 219
kSerial_6O2, 210-212	initUSART1, 220
kSerial_7E1, 210–212	receiveUSART1, 220
kSerial_7E2, 210–212	releaseUSART1, 220
kSerial_7N1, 210-212	transmitUSART1, 221
kSerial_7N2, 210-212	USART2, 57
kSerial_7O1, 210-212	available, 58
kSerial_7O2, 210-212	flush, 58
kSerial_8E1, 210–212	peek, 58
kSerial_8E2, 210-212	read, 58
kSerial_8N1, 210-212	start, 58
kSerial_8N2, 210-212	stop, 59
kSerial_801, 210–212	write, 59, 60
kSerial_8O2, 210–212	USART2.h, 222
<del>-</del> ·	•

kSerial_5E1, 224–226	kSerial_7N2, 231-233
kSerial_5E2, 224–226	kSerial_701, 231-233
kSerial_5N1, 224–226	kSerial_702, 231–233
kSerial_5N2, 224–226	kSerial_8E1, 231-233
kSerial_5O1, 224-226	kSerial_8E2, 231–233
kSerial_5O2, 224-226	kSerial_8N1, 231-233
kSerial_6E1, 224-226	kSerial_8N2, 231-233
kSerial_6E2, 224-226	kSerial_8O1, 231-233
kSerial_6N1, 224-226	kSerial_8O2, 231-233
kSerial_6N2, 224-226	UsartSerialConfiguration, 231
kSerial 601, 224-226	USART3Minimal.h, 233
kSerial 602, 224–226	initUSART3, 234
kSerial 7E1, 224–226	receiveUSART3, 234
kSerial_7E2, 224–226	releaseUSART3, 234
kSerial 7N1, 224–226	transmitUSART3, 235
kSerial 7N2, 224–226	UsartSerialConfiguration
kSerial 701, 224–226	USART0.h, 210
kSerial 702, 224–226	USART1.h, 217
kSerial_8E1, 224–226	USART2.h, 224
kSerial 8E2, 224–226	USART3.h, 231
kSerial 8N1, 224–226	
kSerial_8N2, 224–226	write
kSerial_801, 224–226	I2cLcd, 78
kSerial 802, 224–226	Serial0, 107, 108
UsartSerialConfiguration, 224	Serial1, 122, 123
USART2Minimal.h, 226	Serial2, 138, 139
initUSART2, 227	Serial3, 153, 154
	USART0, 52, 53
receiveUSART2, 227	USART1, 55, 56
releaseUSART2, 227	USART2, 59, 60
transmitUSART2, 228	USART3, 62, 63
USART3, 60	Writer, 164, 165
available, 61	writeAsync
flush, 61	I2cMaster, 36, 37
peek, 61	writeGpioPinDigital
read, 62	GpioPinMacros.h, 182
start, 62	writeGpioPinDigitalV
stop, 62	GpioPinMacros.h, 184
write, 62, 63	writeGpioPinPwm
USART3.h, 229	Pwm.h, 195
kSerial_5E1, 231–233	writeGpioPinPwmV
kSerial_5E2, 231–233	Pwm.h, 199
kSerial_5N1, 231–233	writeSync
kSerial_5N2, 231–233	I2cMaster, 38, 39
kSerial_5O1, 231-233	Writer, 157
kSerial_5O2, 231-233	IntegerOutputBase, 159
kSerial_6E1, 231–233	kBin, 159
kSerial_6E2, 231-233	kDec, 159
kSerial_6N1, 231-233	kHex, 159
kSerial_6N2, 231-233	kOct, 159
kSerial_6O1, 231-233	print, 159–161
kSerial_6O2, 231-233	println, 161–164
kSerial_7E1, 231–233	write, 164, 165
kSerial_7E2, 231–233	Writer.h, 236
<del>-</del> :	willouin, 200
kSerial_7N1, 231–233	