

AVRTools

Generated by Doxygen 1.8.9.1

Sat Sep 5 2015 18:07:54

Contents

1	AVRTools: A Library for the AVR ATmega328 and ATmega2560 Microcontrollers	1
1.1	Overview	1
1.2	Audience	1
1.3	AVRTools is not...	2
1.4	Quick Tour of AVRTools	2
1.4.1	Foundational Elements and Concepts	3
1.4.2	What you need to know about pin name macros	3
1.4.3	The core modules	4
1.4.3.1	System initialization module	5
1.4.3.2	System clock module	5
1.4.3.3	Analog-to-Digital module	5
1.4.3.4	PWM module	5
1.4.3.5	Minimal USART modules	6
1.4.3.6	ABI module	6
1.4.3.7	New module	6
1.5	Sample start up code using AVRTools	6
1.6	Advanced modules	7
1.7	Questions	7
2	Advanced Features	9
2.1	Advanced serial (USART) module	9
2.2	Memory utilities module	10
2.3	Simple delays module	10
2.4	I2C modules	10
2.4.1	I2C Master module	11
2.4.2	I2C Slave module	11
2.5	I2C-based LCD module	12
2.6	GPIO pin variables	12
2.6.1	Example using GPIO pin macros	13

2.6.2	Example using GPIO pin variables	14
3	FAQ	15
3.1	Frequently Asked Questions	15
3.2	Can AVRTools be installed as an Arduino IDE Library?	15
3.3	Why can't I assign pins like pPin01 to a variable?	15
3.4	Why is there a setGpioPinHigh() macro and a _setGpioPinHigh() macro?	15
3.5	_setGpioPinHigh() is defined with 8 arguments, but called with 1: how can that work?	16
3.6	Why is there a setGpioPinHigh() macro and a setGpioPinHighV() function?	16
4	Namespace Index	17
4.1	Namespace List	17
5	Hierarchical Index	19
5.1	Class Hierarchy	19
6	Class Index	21
6.1	Class List	21
7	File Index	23
7.1	File List	23
8	Namespace Documentation	25
8.1	I2cMaster Namespace Reference	25
8.1.1	Detailed Description	26
8.1.2	Enumeration Type Documentation	27
8.1.2.1	I2cBusSpeed	27
8.1.2.2	I2cPullups	27
8.1.2.3	I2cSendErrorCodes	27
8.1.2.4	I2cStatusCodes	28
8.1.3	Function Documentation	28
8.1.3.1	busy	28
8.1.3.2	pullups	28
8.1.3.3	readAsync	28
8.1.3.4	readAsync	29
8.1.3.5	readSync	30
8.1.3.6	readSync	30
8.1.3.7	start	30
8.1.3.8	stop	30

8.1.3.9	writeAsync	31
8.1.3.10	writeAsync	31
8.1.3.11	writeAsync	31
8.1.3.12	writeAsync	32
8.1.3.13	writeSync	32
8.1.3.14	writeSync	33
8.1.3.15	writeSync	33
8.1.3.16	writeSync	33
8.2	I2cSlave Namespace Reference	34
8.2.1	Detailed Description	34
8.2.2	Enumeration Type Documentation	35
8.2.2.1	I2cBusSpeed	35
8.2.2.2	I2cPullups	35
8.2.2.3	I2cStatusCodes	35
8.2.3	Function Documentation	35
8.2.3.1	busy	35
8.2.3.2	processI2cMessage	35
8.2.3.3	pullups	36
8.2.3.4	start	36
8.2.3.5	stop	37
8.3	MemUtils Namespace Reference	37
8.3.1	Detailed Description	37
8.3.2	Function Documentation	37
8.3.2.1	freeRam	37
8.3.2.2	freeRamQuickEstimate	37
8.4	USART0 Namespace Reference	37
8.4.1	Detailed Description	38
8.4.2	Function Documentation	38
8.4.2.1	available	38
8.4.2.2	flush	38
8.4.2.3	peek	39
8.4.2.4	read	39
8.4.2.5	start	39
8.4.2.6	stop	39
8.4.2.7	write	39
8.4.2.8	write	40
8.4.2.9	write	40

8.4.2.10	write	40
8.5	USART1 Namespace Reference	40
8.5.1	Detailed Description	41
8.5.2	Function Documentation	41
8.5.2.1	available	41
8.5.2.2	flush	41
8.5.2.3	peek	42
8.5.2.4	read	42
8.5.2.5	start	42
8.5.2.6	stop	42
8.5.2.7	write	42
8.5.2.8	write	43
8.5.2.9	write	43
8.5.2.10	write	43
8.6	USART2 Namespace Reference	43
8.6.1	Detailed Description	44
8.6.2	Function Documentation	44
8.6.2.1	available	44
8.6.2.2	flush	44
8.6.2.3	peek	45
8.6.2.4	read	45
8.6.2.5	start	45
8.6.2.6	stop	45
8.6.2.7	write	45
8.6.2.8	write	46
8.6.2.9	write	46
8.6.2.10	write	46
8.7	USART3 Namespace Reference	46
8.7.1	Detailed Description	47
8.7.2	Function Documentation	47
8.7.2.1	available	47
8.7.2.2	flush	47
8.7.2.3	peek	48
8.7.2.4	read	48
8.7.2.5	start	48
8.7.2.6	stop	48
8.7.2.7	write	48

8.7.2.8	write	49
8.7.2.9	write	49
8.7.2.10	write	49
9	Class Documentation	51
9.1	GpioPinVariable Class Reference	51
9.1.1	Detailed Description	51
9.2	I2cLcd Class Reference	52
9.2.1	Detailed Description	54
9.2.2	Member Enumeration Documentation	55
9.2.2.1	anonymous enum	55
9.2.2.2	anonymous enum	55
9.2.3	Member Function Documentation	55
9.2.3.1	command	55
9.2.3.2	displayBottomRow	55
9.2.3.3	displayTopRow	55
9.2.3.4	init	56
9.2.3.5	readButtons	56
9.2.3.6	setBacklight	56
9.2.3.7	setCursor	56
9.2.3.8	write	56
9.2.3.9	write	57
9.2.3.10	write	57
9.2.3.11	write	57
9.3	Reader Class Reference	58
9.3.1	Detailed Description	59
9.3.2	Member Function Documentation	59
9.3.2.1	available	59
9.3.2.2	find	60
9.3.2.3	find	60
9.3.2.4	findUntil	60
9.3.2.5	findUntil	60
9.3.2.6	peek	61
9.3.2.7	read	61
9.3.2.8	readBytes	61
9.3.2.9	readBytes	61
9.3.2.10	readBytesUntil	62

9.3.2.11	readBytesUntil	62
9.3.2.12	readFloat	62
9.3.2.13	readFloat	63
9.3.2.14	readLine	63
9.3.2.15	readLong	63
9.3.2.16	readLong	63
9.3.2.17	setTimeout	64
9.4	RingBuffer Class Reference	64
9.4.1	Detailed Description	65
9.4.2	Constructor & Destructor Documentation	65
9.4.2.1	RingBuffer	65
9.4.3	Member Function Documentation	65
9.4.3.1	isEmpty	65
9.4.3.2	isFull	65
9.4.3.3	isEmpty	65
9.4.3.4	isNotFull	66
9.4.3.5	peek	66
9.4.3.6	pull	66
9.4.3.7	push	66
9.5	RingBufferT< T, N, SIZE > Class Template Reference	66
9.5.1	Detailed Description	67
9.5.2	Member Function Documentation	67
9.5.2.1	discardFromFront	67
9.5.2.2	isEmpty	68
9.5.2.3	isFull	68
9.5.2.4	isEmpty	68
9.5.2.5	isNotFull	68
9.5.2.6	peek	68
9.5.2.7	pull	69
9.5.2.8	push	69
9.6	Serial0 Class Reference	69
9.6.1	Detailed Description	70
9.6.2	Member Function Documentation	71
9.6.2.1	available	71
9.6.2.2	peek	71
9.6.2.3	read	71
9.6.2.4	start	71

9.6.2.5	stop	72
9.6.2.6	write	72
9.6.2.7	write	72
9.6.2.8	write	72
9.6.2.9	write	73
9.7	Serial1 Class Reference	73
9.7.1	Detailed Description	74
9.7.2	Member Function Documentation	75
9.7.2.1	available	75
9.7.2.2	peek	75
9.7.2.3	read	75
9.7.2.4	start	75
9.7.2.5	stop	76
9.7.2.6	write	76
9.7.2.7	write	76
9.7.2.8	write	76
9.7.2.9	write	77
9.8	Serial2 Class Reference	77
9.8.1	Detailed Description	78
9.8.2	Member Function Documentation	79
9.8.2.1	available	79
9.8.2.2	peek	79
9.8.2.3	read	79
9.8.2.4	start	79
9.8.2.5	stop	80
9.8.2.6	write	80
9.8.2.7	write	80
9.8.2.8	write	80
9.8.2.9	write	81
9.9	Serial3 Class Reference	81
9.9.1	Detailed Description	82
9.9.2	Member Function Documentation	83
9.9.2.1	available	83
9.9.2.2	peek	83
9.9.2.3	read	83
9.9.2.4	start	83
9.9.2.5	stop	84

9.9.2.6	write	84
9.9.2.7	write	84
9.9.2.8	write	84
9.9.2.9	write	85
9.10	Writer Class Reference	85
9.10.1	Detailed Description	87
9.10.2	Member Enumeration Documentation	87
9.10.2.1	IntegerOutputBase	87
9.10.3	Member Function Documentation	87
9.10.3.1	print	87
9.10.3.2	print	87
9.10.3.3	print	88
9.10.3.4	print	88
9.10.3.5	print	88
9.10.3.6	print	89
9.10.3.7	print	89
9.10.3.8	print	89
9.10.3.9	println	90
9.10.3.10	println	90
9.10.3.11	println	90
9.10.3.12	println	90
9.10.3.13	println	91
9.10.3.14	println	91
9.10.3.15	println	91
9.10.3.16	println	91
9.10.3.17	println	92
9.10.3.18	write	92
9.10.3.19	write	92
9.10.3.20	write	92
9.10.3.21	write	93
10	File Documentation	95
10.1	abi.h File Reference	95
10.1.1	Detailed Description	95
10.2	Analog2Digital.h File Reference	96
10.2.1	Detailed Description	97
10.2.2	Macro Definition Documentation	97

10.2.2.1	readGpioPinAnalog	97
10.2.3	Enumeration Type Documentation	97
10.2.3.1	A2DVoltageReference	97
10.2.4	Function Documentation	98
10.2.4.1	initA2D	98
10.2.4.2	readA2D	98
10.2.4.3	readGpioPinAnalogV	98
10.2.4.4	setA2DVoltageReference	99
10.2.4.5	setA2DVoltageReference11V	99
10.2.4.6	setA2DVoltageReference256V	99
10.2.4.7	setA2DVoltageReferenceAREF	99
10.2.4.8	setA2DVoltageReferenceAVCC	99
10.3	ArduinoMegaPins.h File Reference	99
10.3.1	Detailed Description	100
10.4	ArduinoPins.h File Reference	101
10.4.1	Detailed Description	101
10.5	ArduinoUnoPins.h File Reference	101
10.5.1	Detailed Description	102
10.6	GpioPinMacros.h File Reference	102
10.6.1	Detailed Description	105
10.6.2	Macro Definition Documentation	105
10.6.2.1	getGpioADC	105
10.6.2.2	getGpioCOM	105
10.6.2.3	getGpioDDR	106
10.6.2.4	getGpioMASK	106
10.6.2.5	getGpioOCR	106
10.6.2.6	getGpioPIN	106
10.6.2.7	getGpioPORT	106
10.6.2.8	getGpioTCCR	107
10.6.2.9	GpioPin	107
10.6.2.10	GpioPinAnalog	107
10.6.2.11	GpioPinPwm	107
10.6.2.12	makeGpioVarFromGpioPin	107
10.6.2.13	makeGpioVarFromGpioPinAnalog	108
10.6.2.14	makeGpioVarFromGpioPinPwm	108
10.6.2.15	readGpioPinDigital	108
10.6.2.16	setGpioPinHigh	108

10.6.2.17	setGpioPinLow	108
10.6.2.18	setGpioPinModeInput	109
10.6.2.19	setGpioPinModeInputPullup	109
10.6.2.20	setGpioPinModeOutput	109
10.6.2.21	writeGpioPinDigital	109
10.6.3	Enumeration Type Documentation	109
10.6.3.1	anonymous enum	109
10.6.4	Function Documentation	109
10.6.4.1	readGpioPinDigitalV	109
10.6.4.2	setGpioPinHighV	110
10.6.4.3	setGpioPinLowV	110
10.6.4.4	setGpioPinModeInputPullupV	110
10.6.4.5	setGpioPinModeInputV	110
10.6.4.6	setGpioPinModeOutputV	110
10.6.4.7	writeGpioPinDigitalV	110
10.7	I2cLcd.h File Reference	111
10.7.1	Detailed Description	111
10.8	I2cMaster.h File Reference	111
10.8.1	Detailed Description	114
10.9	I2cSlave.h File Reference	114
10.9.1	Detailed Description	116
10.10	InitSystem.h File Reference	116
10.10.1	Detailed Description	116
10.10.2	Function Documentation	116
10.10.2.1	initSystem	117
10.11	MemUtils.h File Reference	117
10.11.1	Detailed Description	117
10.12	new.h File Reference	117
10.12.1	Detailed Description	118
10.13	Pwm.h File Reference	118
10.13.1	Detailed Description	120
10.13.2	Macro Definition Documentation	120
10.13.2.1	writeGpioPinPwm	121
10.13.3	Function Documentation	121
10.13.3.1	clearTimer0	121
10.13.3.2	clearTimer1	121
10.13.3.3	clearTimer2	121

10.13.3.4 clearTimer3	122
10.13.3.5 clearTimer4	122
10.13.3.6 clearTimer5	122
10.13.3.7 initPwmTimer0	122
10.13.3.8 initPwmTimer1	123
10.13.3.9 initPwmTimer2	123
10.13.3.10 initPwmTimer3	123
10.13.3.11 initPwmTimer4	123
10.13.3.12 initPwmTimer5	124
10.13.3.13 writeGpioPinPwmV	124
10.14 Reader.h File Reference	124
10.14.1 Detailed Description	125
10.15 RingBuffer.h File Reference	125
10.15.1 Detailed Description	126
10.16 RingBufferT.h File Reference	126
10.16.1 Detailed Description	127
10.17 SimpleDelays.h File Reference	127
10.17.1 Detailed Description	128
10.17.2 Function Documentation	128
10.17.2.1 delayQuartersOfMicroSeconds	128
10.17.2.2 delayTenthsOfSeconds	129
10.17.2.3 delayWholeMilliSeconds	129
10.18 SystemClock.h File Reference	129
10.18.1 Detailed Description	130
10.18.2 Function Documentation	130
10.18.2.1 delay	130
10.18.2.2 delayMicroseconds	130
10.18.2.3 delayMilliseconds	130
10.18.2.4 initSystemClock	131
10.18.2.5 micros	131
10.18.2.6 millis	131
10.19 USART0.h File Reference	131
10.19.1 Detailed Description	134
10.19.2 Enumeration Type Documentation	134
10.19.2.1 UsartSerialConfiguration	134
10.20 USART1.h File Reference	137
10.20.1 Detailed Description	139

10.20.2 Enumeration Type Documentation	139
10.20.2.1 UsartSerialConfiguration	139
10.21 USART2.h File Reference	142
10.21.1 Detailed Description	144
10.21.2 Enumeration Type Documentation	144
10.21.2.1 UsartSerialConfiguration	144
10.22 USART3.h File Reference	147
10.22.1 Detailed Description	149
10.22.2 Enumeration Type Documentation	149
10.22.2.1 UsartSerialConfiguration	149
10.23 USARTMinimal.h File Reference	152
10.23.1 Detailed Description	153
10.23.2 Function Documentation	153
10.23.2.1 initUSART0	153
10.23.2.2 initUSART1	153
10.23.2.3 initUSART2	154
10.23.2.4 initUSART3	154
10.23.2.5 receiveUSART0	154
10.23.2.6 receiveUSART1	155
10.23.2.7 receiveUSART2	155
10.23.2.8 receiveUSART3	155
10.23.2.9 releaseUSART0	155
10.23.2.10 releaseUSART1	156
10.23.2.11 releaseUSART2	156
10.23.2.12 releaseUSART3	156
10.23.2.13 transmitUSART0	156
10.23.2.14 transmitUSART0	156
10.23.2.15 transmitUSART1	157
10.23.2.16 transmitUSART1	157
10.23.2.17 transmitUSART2	157
10.23.2.18 transmitUSART2	157
10.23.2.19 transmitUSART3	158
10.23.2.20 transmitUSART3	158
10.24 Writer.h File Reference	158
10.24.1 Detailed Description	159

Chapter 1

AVRTools: A Library for the AVR ATmega328 and ATmega2560 Microcontrollers

1.1 Overview

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

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

Like the Arduino libraries, AVRTools allows you to refer to pins on an Arduino via simple names such as `pPin07` for digital pin 7 or `pPinA03` for analog pin 3. Unlike the Arduino libraries, these names are pure macros so that `setGpioPinHigh(pPin12)` always translates directly into `PORTB |= (1<<4)` on an Arduino Uno. Similar macros are available for conveniently naming any pin on an ATmega328 or ATmega2560 and provide easy and efficient access to all the functionality available on that pin (digital I/O, analog-to-digital conversion, PWM, etc). AVRTools provides functions to access the primary 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 functions 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:

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

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

- You wish you could use a function-like syntax to switch input/output mode, read a pin, or set a pin high or low but still have the compiler generate 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 assumes the microcontrollers are running at 16 MHz. I believe the only place this matters is in clock, timing, and delay related functions, but I haven't tested the code at any CPU speed other than 16 MHz.

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

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

1.4 Quick Tour of AVRTools

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

1.4.1 Foundational Elements and Concepts

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

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

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

- `setGpioPinModeOutput(pin)` Enable the corresponding DDRn bit
- `setGpioPinModeInput(pin)` Clear the corresponding DDRn bit
- `setGpioPinModeInputPullup(pin)` Clear the corresponding DDRn and PORTn bits
- `readGpioPinDigital(pin)` Return the corresponding PINn bit
- `writeGpioPinDigital(pin, value)` Write a 0 or 1 to the corresponding PORTn bit
- `setGpioPinHigh(pin)` Set the corresponding PORTn bit
- `setGpioPinLow(pin)` Clear the corresponding PORTn bit
- `readGpioPinAnalog(pin)` Read an analog value from the corresponding ADC channel
- `writeGpioPinPwm(pin, value)` Set the corresponding PWM output level for that pin

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

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

setGpioPinModeOutput( THE_RED_LED );
setGpioPinLow( THE_RED_LED );

setGpioPinModeOutput( THE_GRN_LED );
setGpioPinLow( THE_GRN_LED );

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

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

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

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

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

1.4.3 The core modules

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

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

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

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

Brief descriptions of these modules follow.

1.4.3.1 System initialization module

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

```
void initSystem();
```

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

1.4.3.2 System clock module

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

1.4.3.3 Analog-to-Digital module

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

```
void initA2D();
void turnOffA2D();
readGpioPinAnalog( 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:

```
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 either `initPwmTimer1()` or `initPwmTimer2()` 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.

1.4.3.5 Minimal USART modules

This module provides a simple and minimal means of reading and writing from the USARTs available on the ATmega328 and ATmega2560. To employ this functionality, you must include the header file `USARTMinimal.h` and link against the file `USARTMinimal.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 USART capability on `USART0`, 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. 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"

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

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 modules include both master and slave I2C modules (transmitting and receiving via interrupts), a module for driving an LCD display via I2C, a module for reporting memory utilization, and a module for more advanced serial input and output of various numerical types and strings. Information on these modules can be found in the [Advanced Features](#) sections of the documentation.

1.7 Questions

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

Chapter 2

Advanced Features

The AVRTools library includes four more advanced features:

- [Advanced serial \(USART\) module](#)
- [Memory utilities module](#)
- [Simple delays module](#)
- [I2C modules](#)
- [I2C-based LCD 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 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 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.

2.3 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.

2.4 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.

2.4.1 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.

2.4.2 I2C Slave module

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

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

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

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

2.5 I2C-based LCD module

The [I2C-base 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 associated with such devices.

Note

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

2.6 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 specialize 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, hiding the pins from the rest of the application. Treating the pins as macro constants usually works well in such situations. However, there do sometimes arise situations in which it would be convenient to be able to assign GPIO pins to variables and manipulate GPIO pins via those variables.

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

The first reason is that functions that manipulate AVR I/O registers via variables are inherently slower than those that manipulate them as constants. When using the GPIO pin macros, most operations map directly to `in` and `out` AVR assembler instructions. However, due to the limitations of 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 just 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 assign and passed to functions as needed. To use these GPIO pin variables, there are special function analogs of the pin manipulation macros. These have the same names as the pin manipulation macros, except with a "V" appended:

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

Note

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

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

2.6.1 Example using GPIO pin macros

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

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

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

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

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

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

```

delayMilliseconds( 2000 );

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

while ( 1 )
{
    delayMilliseconds( 1000 );

    setGpioPinLow( pRed );
    setGpioPinHigh( pGreen );

    delayMilliseconds( 1000 );

    setGpioPinLow( pGreen );
    setGpioPinHigh( pYellow );

    delayMilliseconds( 1000 );

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

```

2.6.2 Example using GPIO pin variables

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

```

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

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

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

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

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

    delayMilliseconds( 2000 );

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

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

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

```

Chapter 3

FAQ

3.1 Frequently Asked Questions

- [Can AVRTools be installed as an Arduino IDE Library?](#)
- [Why can't I assign pins like pPin01 to a variable?](#)
- [Why 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.

3.4 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.

3.5 `_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()`.

3.6 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:

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	25
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	34
MemUtils	A namespace providing encapsulation for functions that report the available memory in SRAM . . .	37
USART0	This namespace bundles a high-level buffered interface to the USART0 hardware. It provides logical cohesion and prevents namespace collisions	37
USART1	This namespace bundles a high-level buffered interface to the USART1 hardware. It provides logical cohesion and prevents namespace collisions	40
USART2	This namespace bundles a high-level buffered interface to the USART2 hardware. It provides logical cohesion and prevents namespace collisions	43
USART3	This namespace bundles a high-level buffered interface to the USART3 hardware. It provides logical cohesion and prevents namespace collisions	46

Chapter 5

Hierarchical Index

5.1 Class Hierarchy

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

GpioPinVariable	51
Reader	58
Serial0	69
Serial1	73
Serial2	77
Serial3	81
RingBuffer	64
RingBufferT< T, N, SIZE >	66
Writer	85
I2cLcd	52
Serial0	69
Serial1	73
Serial2	77
Serial3	81

Chapter 6

Class Index

6.1 Class List

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

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	51
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	52
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)	58
RingBuffer	This class provides an efficient ring buffer implementation for storing bytes. Ring buffers are particularly useful for memory constrained microcontrollers such as the ATmega328 and ATmega2650. For maximum efficiency, this class is focused on the storage of bytes, providing a single code base that is shared by all instances of this class	64
RingBufferT< T, N, SIZE >	Template-based ring buffer class that can store different kinds of objects in buffers of whatever size is needed	66
Serial0	Provides a high-end interface to serial communications using USART0	69
Serial1	Provides a high-end interface to serial communications using USART1	73
Serial2	Provides a high-end interface to serial communications using USART2	77
Serial3	Provides a high-end interface to serial communications using USART3	81
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)	85

Chapter 7

File Index

7.1 File List

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

abi.h	This file provides certain functions needed to complete the avr-gcc C++ ABI. You never need to include this file, and you only need to link against abi.cpp if you encounter certain link errors	95
Analog2Digital.h	This file provides functions that access the analog-to-digital conversion capability of the ATmega328 and ATmega2560 microcontrollers	96
ArduinoMegaPins.h	This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file ArduinoPins.h , which in turn includes this file (when compiling for Arduino Uno targets)	99
ArduinoPins.h	This file is the primary one that users should include to access and use the pin name macros	101
ArduinoUnoPins.h	This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file ArduinoPins.h , which in turn includes this file (when compiling for Arduino Uno targets)	101
GpioPinMacros.h	This file contains the primary macros for naming and manipulating GPIO pin names	102
I2cLcd.h	This file defines a class that provides a high-level interface to an LCD offering an I2C interface. The most common variant of this is HD44780U controlled LCD driven via an MCP23017 that offers an I2C interface (such LCDs are available from Adafruit and SparkFun). To use this class you must also use and properly initialize the I2C Master package from I2cMaster.h	111
I2cMaster.h	This file provides functions that interface to the TWI (two-wire serial interface) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. Include this file if you want your application will operate in Master mode as defined in the I2C protocol	111
I2cSlave.h	This file provides functions that interface to the TWI (two-wire serial interface) hardware of the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560), providing a high-level interface to I2C protocol communications. Include this file if you want your application will operate in Slave mode as defined in the I2C protocol	114

InitSystem.h	Include this file to use the functions that initialize the microcontroller to a known, basic state	116
MemUtils.h	This file provides functions that provide information on the available memory in SRAM	117
new.h	This file provides <code>operator new</code> and <code>operator delete</code> . You only need this file if you use <code>new</code> and <code>delete</code> to manage objects on the heap	117
Pwm.h	This file provides functions that access the PWM capability of the ATmega328 and ATmega2560 microcontrollers	118
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	124
RingBuffer.h	This file provides an efficient ring buffer implementation for storing bytes	125
RingBufferT.h	This file provides a very flexible, template-based ring buffer implementation	126
SimpleDelays.h	This file provides simple delay functions that do not involve timers or interrupts. These functions simply execute a series of nested loops with known and precise timing	127
SystemClock.h	Include this file to use the functions that instantiate and access a system clock that counts elapsed milliseconds	129
USART0.h	This file provides functions that offer high-level interfaces to USART0 hardware, which is available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560)	131
USART1.h	This file provides functions that offer high-level interfaces to USART1 hardware, which is available on Arduino Mega (ATmega2560)	137
USART2.h	This file provides functions that offer high-level interfaces to USART2 hardware, which is available on Arduino Mega (ATmega2560)	142
USART3.h	This file provides functions that offer high-level interfaces to USART3 hardware, which is available on Arduino Mega (ATmega2560)	147
USARTMinimal.h	This file provides functions that provide a minimalist interface to the USARTs available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560)	152
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	158

Chapter 8

Namespace Documentation

8.1 I2cMaster Namespace Reference

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

Enumerations

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

Functions

- void `start` (uint8_t speed=`kI2cBusFast`)
Configures the TWI hardware for I2C communications in Master mode. You must call this function before conducting any I2C communications using the functions in this module.
- void `stop` ()
Terminates the I2C communications using the TWI hardware, and disables the TWI interrupts.
- void `pullups` (uint8_t set=`kPullupsOn`)
Sets the state of the internal pullups that are part of the TWI hardware.
- bool `busy` ()
Reports whether the TWI hardware is busy communicating (either transmitting or receiving).
- uint8_t `writeAsync` (uint8_t address, uint8_t registerAddress, volatile uint8_t *status)
Transmit a single register address (a one-byte message) asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).

- `uint8_t writeAsync` (`uint8_t address`, `uint8_t registerAddress`, `uint8_t data`, `volatile uint8_t *status`)
Transmit a single register address and corresponding single byte of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- `uint8_t writeAsync` (`uint8_t address`, `uint8_t registerAddress`, `const char *data`, `volatile uint8_t *status`)
Transmit a single register address and corresponding null-terminated string of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- `uint8_t writeAsync` (`uint8_t address`, `uint8_t registerAddress`, `uint8_t *data`, `uint8_t numberBytes`, `volatile uint8_t *status`)
Transmit a single register address and corresponding buffer of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- `uint8_t readAsync` (`uint8_t address`, `uint8_t numberBytes`, `volatile uint8_t *destination`, `volatile uint8_t *bytesRead`, `volatile uint8_t *status`)
Request to read data from a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports `kI2cCompletedOk`, 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 `kI2cCompletedOk`, 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 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 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 error codes that may occur when you try to write a message.

Enumerator

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

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

Request to read data from a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports `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 counter in which the TWI hardware will asynchronously keep track of how many bytes have been received.
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be updated asynchronously after the function returns by the TWI hardware) values correspond to `I2cStatusCodes`.

Returns

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

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

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

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

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

Returns

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

8.1.3.5 `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 (`kI2cBusFast`).

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

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

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

- `address` the I2C address of the destination device for this message
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., turn off or on).
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be 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 I2cMaster::writeAsync (uint8_t address, uint8_t registerAddress, uint8_t data, volatile uint8_t * status)`

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

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

- `address` the I2C address of the destination device for this message
- `registerAddress` in device-centric terms, the register address on the destination device; think of it as a one-byte instruction to the destination device telling it to do something (e.g., set the volume level).
- `data` a single byte of data serving as a parameter to the register address (e.g., the volume level to set).
- `status` a pointer to a byte-size location in which the communications status of this message will be reported (volatile because the value will be 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 I2cMaster::writeAsync (uint8_t address, uint8_t registerAddress, const char * data, volatile uint8_t * status)`

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

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

- `address` the I2C address of the destination device for this message

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

Returns

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

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

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

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

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

Returns

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

8.1.3.13 `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

- enum `I2cBusSpeed` { `kI2cBusSlow`, `kI2cBusFast` }
This enum lists I2C bus speed configurations.
- enum `I2cStatusCodes` {
 `kI2cCompletedOk` = 0x00, `kI2cError` = 0x01, `kI2cTxPartial` = 0x02, `kI2cRxOverflow` = 0x04,
 `kI2cInProgress` = 0x06 }
This enum lists I2C status codes reported by the various transmit functions.
- enum `I2cPullups` { `kPullupsOff`, `kPullupsOn` }
This enum lists the options for controlling the built-in pullups in the TWI hardware.

Functions

- `uint8_t processI2cMessage` (`uint8_t *buffer`, `uint8_t len`)
This function must be defined by the user. It is called by the TWI interrupt function installed as part of `I2cSlave.cpp` whenever it receives a message from the Master. The user should implement this function to respond to the data in the buffer, taking actions and as appropriate returning data to the buffer (for asynchronous transmission to the Master).
- `void start` (`uint8_t ownAddress`, `uint8_t speed=kI2cBusFast`, `bool answerGeneralCall=false`)
Configures the TWI hardware for I2C communications in Slave mode. You must call this function before conducting any I2C communications using the functions in this module.
- `void stop` ()
Terminates the I2C communications using the TWI hardware, and disables the TWI interrupts.
- `void pullups` (`uint8_t set=kPullupsOn`)
Sets the state of the internal pullups that are part of the TWI hardware.
- `bool busy` ()
Reports whether the TWI hardware is busy communicating (either transmitting or receiving).

8.2.1 Detailed Description

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

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

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

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

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

8.2.2 Enumeration Type Documentation

8.2.2.1 enum I2cSlave::I2cBusSpeed

This enum lists I2C bus speed configurations.

Enumerator

kI2cBusSlow I2C slow (standard) mode: 100 KHz.

kI2cBusFast I2C fast mode: 400 KHz.

8.2.2.2 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

kI2cCompletedOk I2C communications completed with no error.

kI2cError I2C communications encountered an error.

kI2cTxPartial I2C Master terminated transmission before all data were sent.

kI2cRxOverflow Received a message larger than can be held in the receive buffer.

kI2cInProgress I2C communications on this message still in progress.

8.2.3 Function Documentation

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

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

The user should implement this function to do the following:

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

Note

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

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

Returns

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

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

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

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

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

8.2.3.5 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 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.

8.3.1 Detailed Description

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

8.3.2 Function Documentation

8.3.2.1 unsigned int MemUtils::freeRam ()

Get the number of free bytes remaining in SRAM.

Returns

The number of free bytes remaining in SRAM.

8.3.2.2 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.

8.4 USART0 Namespace Reference

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

Functions

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

8.4.1 Detailed Description

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

8.4.2 Function Documentation

8.4.2.1 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;

8.4.2.2 void USART0::flush ()

Flush transmit buffer.

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

8.4.2.3 `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;

8.4.2.4 `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;

8.4.2.5 `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.

8.4.2.6 `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\(\)](#).

8.4.2.7 `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.

8.4.2.8 `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.

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

Returns

the number of bytes written into the output buffer.

8.4.2.9 `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.

8.4.2.10 `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.

8.5 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..

8.5.1 Detailed Description

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

8.5.2 Function Documentation

8.5.2.1 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;

8.5.2.2 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.

8.5.2.3 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;

8.5.2.4 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;

8.5.2.5 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.

8.5.2.6 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\(\)](#).

8.5.2.7 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.

8.5.2.8 `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.

8.5.2.9 `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.

8.5.2.10 `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.

8.6 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..

8.6.1 Detailed Description

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

8.6.2 Function Documentation

8.6.2.1 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;

8.6.2.2 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.

8.6.2.3 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;

8.6.2.4 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;

8.6.2.5 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.

8.6.2.6 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\(\)](#).

8.6.2.7 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.

8.6.2.8 `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.

8.6.2.9 `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.

8.6.2.10 `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.

8.7 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..

8.7.1 Detailed Description

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

8.7.2 Function Documentation

8.7.2.1 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;

8.7.2.2 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.

8.7.2.3 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;

8.7.2.4 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;

8.7.2.5 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.

8.7.2.6 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\(\)](#).

8.7.2.7 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.

8.7.2.8 `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.

8.7.2.9 `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.

8.7.2.10 `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

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

Returns

the number of bytes written into the output buffer.

Chapter 9

Class Documentation

9.1 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).

9.1.1 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 variables of type [GpioPinVariable](#): [makeGpioVarFromGpioPin\(\)](#), [makeGpioVarFromGpioPinAnalog\(\)](#), and [makeGpioVarFromGpioPinPwm\(\)](#). These are used like this:

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

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

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

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

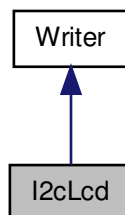
- [GpioPinMacros.h](#)

9.2 I2cLcd Class Reference

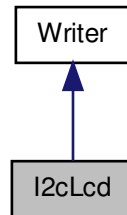
This class provides a high-level interface via I2C to an LCD such as those offered by AdaFruit and SparkFun. Specifically, it communicates via I2C with an MCP23017 that drives an HD44780U controlling an LCD. It also lets you detect button presses on the 5-button keypad generally associated with such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

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

Inheritance diagram for I2cLcd:



Collaboration diagram for I2cLcd:



Public Types

- enum {
 [kButton_Select](#), [kButton_Right](#), [kButton_Down](#), [kButton_Up](#),
 [kButton_Left](#) }
- enum {
 [kBacklight_Red](#), [kBacklight_Yellow](#), [kBacklight_Green](#), [kBacklight_Teal](#),
 [kBacklight_Blue](#), [kBacklight_Violet](#), [kBacklight_White](#) }

Public Member Functions

- [I2cLcd](#) ()
 Constructor simply initializes some internal bookkeeping.
- int [init](#) ()
 Initialize the [I2cLcd](#) object. This must be called before using the [I2cLcd](#), or calling any of the other member functions. The I2C system must be initialized before calling this function (by calling [I2cMaster::start\(\)](#) from [I2cMaster.h](#)).
- void [clear](#) ()
 Clear the display (all rows, all columns).
- void [home](#) ()
 Move the cursor home (the top row, left column).
- void [displayTopRow](#) (const char *str)
 Display a C-string on the top row.
- void [displayBottomRow](#) (const char *str)
 Display a C-string on the bottom row.
- void [clearTopRow](#) ()
 Clear the top row.
- void [clearBottomRow](#) ()
 Clear the bottom row.
- void [displayOff](#) ()
 Turn the display off.
- void [displayOn](#) ()
 Turn the display on.

- void `blinkOff` ()
Do not blink the cursor.
- void `blinkOn` ()
Blink the cursor.
- void `cursorOff` ()
Hide the cursor.
- void `cursorOn` ()
Display the cursor.
- void `scrollDisplayLeft` ()
Scroll the display to the left.
- void `scrollDisplayRight` ()
Scroll the display to the right.
- void `autoscrollOn` ()
Turn on automatic scrolling of the display.
- void `autoscrollOff` ()
Turn off automatic scrolling of the display.
- void `setCursor` (uint8_t row, uint8_t col)
Move the cursor the a particular row and column.
- int `setBacklight` (uint8_t color)
Set the backlight to a given color. Set a black-and-white LCD display to White if you want to have a backlight.
- void `command` (uint8_t cmd)
Pass a command to the LCD.
- uint8_t `readButtons` ()
Read the state of the buttons associated with the LCD display.
- virtual size_t `write` (char c)
Write a single character to the LCD at the current cursor location. This implements the pure virtual function `Writer::write(char c)`.
- virtual size_t `write` (const char *str)
Write a C-string to the LCD at the current cursor location. This implements the pure virtual function `Writer::write(const char str)`.*
- virtual size_t `write` (const char *buffer, size_t size)
Write a given number of characters from a buffer to the LCD at the current cursor location. This implements the pure virtual function `Writer::write(const char buffer, size_t size)`.*
- virtual size_t `write` (const uint8_t *buffer, size_t size)
Write a given number of bytes from a buffer to the LCD at the current cursor location. This implements the pure virtual function `Writer::write(const uint8_t buffer, size_t size)`.*
- virtual void `flush` ()
This function does nothing. It simply implements the pure virtual function `Writer::flush()`.

9.2.1 Detailed Description

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

This class derives from `Writer`, allowing you to write to the LCD much as it if were a serial device.

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

9.2.2 Member Enumeration Documentation

9.2.2.1 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

9.2.2.2 anonymous enum

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

Enumerator

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

9.2.3 Member Function Documentation

9.2.3.1 void I2cLcd::command (uint8_t *cmd*)

Pass a command to the LCD.

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

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

Display a C-string on the bottom row.

- *str* the C-string to display.

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

Display a C-string on the top row.

- *str* the C-string to display.

9.2.3.4 `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.

9.2.3.5 `uint8_t I2cLcd::readButtons ()`

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

Returns

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

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

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

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

9.2.3.7 `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).

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

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

- `the` character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

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

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

- `str` the C-string to be written.

Returns

the number of bytes written.

Implements [Writer](#).

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

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

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

Returns

the number of bytes written.

Implements [Writer](#).

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

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

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

Returns

the number of bytes written.

Implements [Writer](#).

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

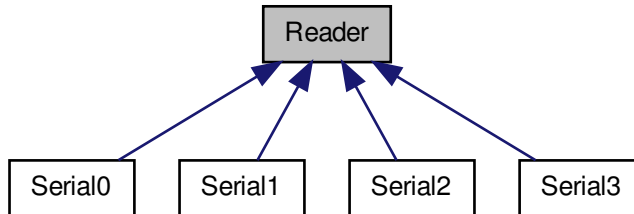
- [I2cLcd.h](#)
- [I2cLcd.cpp](#)

9.3 Reader Class Reference

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

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

Inheritance diagram for Reader:



Public Member Functions

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

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

- bool [readFloat](#) (float *result, char skipChar)

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

- size_t [readBytes](#) (char *buffer, size_t length)

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

- size_t [readBytesUntil](#) (char terminator, char *buffer, size_t length)

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

- size_t [readBytes](#) (uint8_t *buffer, size_t length)

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

- size_t [readBytesUntil](#) (uint8_t terminator, uint8_t *buffer, size_t length)

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

- size_t [readLine](#) (char *buffer, size_t length)

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

- void [consumeWhiteSpace](#) ()

Consumes whitespace characters until the first non-whitespace character is encountered or the function times out.

9.3.1 Detailed Description

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

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

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

Note

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

9.3.2 Member Function Documentation

9.3.2.1 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](#).

9.3.2.2 `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.

9.3.2.3 `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.

9.3.2.4 `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.

9.3.2.5 `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.

9.3.2.6 `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](#).

9.3.2.7 `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](#).

9.3.2.8 `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).

9.3.2.9 `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).

9.3.2.10 `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).

9.3.2.11 `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).

9.3.2.12 `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.

9.3.2.13 `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.

9.3.2.14 `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).

9.3.2.15 `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.

9.3.2.16 `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.

9.3.2.17 `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](#)

9.4 RingBuffer Class Reference

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

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

Public Member Functions

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

9.4.1 Detailed Description

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

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

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

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

9.4.2 Constructor & Destructor Documentation

9.4.2.1 `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.

9.4.3 Member Function Documentation

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

Determine if the buffer is empty .

Returns

true if the buffer is empty; false if not.

9.4.3.2 `bool RingBuffer::isFull ()`

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

Returns

true if the buffer is full; false if not.

9.4.3.3 `bool RingBuffer::isNotEmpty () [inline]`

Determine if the buffer is not empty.

Returns

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

9.4.3.4 `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.

9.4.3.5 `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.

9.4.3.6 `int RingBuffer::pull ()`

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

Returns

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

9.4.3.7 `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](#)

9.5 `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 isEmpty](#) ()
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.

9.5.1 Detailed Description

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

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

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

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

Template Parameters

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

9.5.2 Member Function Documentation

9.5.2.1 `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.

9.5.2.2 `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.

9.5.2.3 `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.

9.5.2.4 `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.

9.5.2.5 `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.

9.5.2.6 `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.

9.5.2.7 `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.

9.5.2.8 `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:

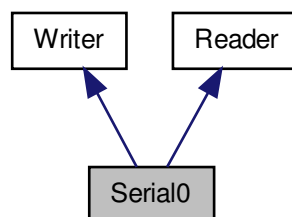
- [RingBufferT.h](#)

9.6 Serial0 Class Reference

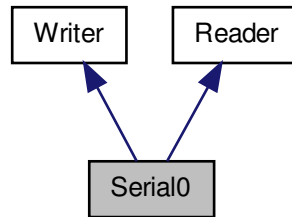
Provides a high-end interface to serial communications using USART0.

```
#include <USART0.h>
```

Inheritance diagram for Serial0:



Collaboration diagram for Serial0:



Public 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\(\)](#).

Additional Inherited Members

9.6.1 Detailed Description

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

The functions in this class are buffered for both input and output and operate using interrupts associated with USART0. This means the write functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART0 hardware. Similarly, data is received asynchronously and placed into the read buffer.

The read and write buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The read buffer, however, will overwrite if it gets full. You must clear the read buffer by actually reading the data regularly when receiving significant amounts of data.

9.6.2 Member Function Documentation

9.6.2.1 `virtual bool Serial0::available () [virtual]`

Determine if data is available in the input stream. This implements the pure virtual function [Reader::available\(\)](#).

Returns

True if data is available in the stream; false if not.

Implements [Reader](#).

9.6.2.2 `virtual int Serial0::peek () [virtual]`

Examine the next byte from the input stream, without removing it. This implements the pure virtual function [Reader::peek\(\)](#).

Returns

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

Implements [Reader](#).

9.6.2.3 `virtual int Serial0::read () [virtual]`

Read and remove the next byte from the input stream. This implements the pure virtual function [Reader::read\(\)](#).

Returns

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

Implements [Reader](#).

9.6.2.4 `void Serial0::start (unsigned long baudRate, UsartSerialConfiguration config = kSerial_8N1) [inline]`

Configure the hardware for two-way serial communications, including turning on associated interrupts. You must call this function before reading from or writing to [Serial0](#) on USART0.

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

9.6.2.5 void Serial0::stop () [inline]

Stops buffered serial communications using [Serial0](#) on USART0 by deconfiguring the hardware and turning off interrupts.

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

If you want to use [Serial0](#) again for communications, you must call [start\(\)](#) again.

9.6.2.6 virtual size_t Serial0::write (char c) [virtual]

Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).

- `c` the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.6.2.7 virtual size_t Serial0::write (const char * str) [virtual]

Write a null-terminated string to the output stream. This implements the pure virtual function [Writer::write\(char* str \)](#).

- `str` the string to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.6.2.8 virtual size_t Serial0::write (const char * buffer, size_t size) [virtual]

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

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

Returns

the number of bytes written.

Implements [Writer](#).

9.6.2.9 `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:

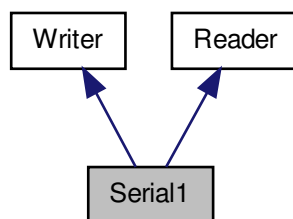
- [USART0.h](#)

9.7 Serial1 Class Reference

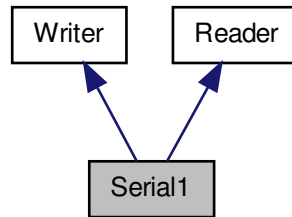
Provides a high-end interface to serial communications using USART1.

```
#include <USART1.h>
```

Inheritance diagram for Serial1:



Collaboration diagram for Serial1:



Public 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\(\)](#).

Additional Inherited Members

9.7.1 Detailed Description

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

The functions in this class are buffered for both input and output and operate using interrupts associated with USART1. This means the write functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART1 hardware. Similarly, data is received asynchronously and placed into the read buffer.

The read and write buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The read buffer, however, will overwrite if it gets full. You must clear the read buffer by actually reading the data regularly when receiving significant amounts of data.

9.7.2 Member Function Documentation

9.7.2.1 `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](#).

9.7.2.2 `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](#).

9.7.2.3 `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](#).

9.7.2.4 `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.

9.7.2.5 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.

9.7.2.6 size_t Serial1::write (char c) [virtual]

Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).

- `c` the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.7.2.7 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](#).

9.7.2.8 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](#).

9.7.2.9 `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:

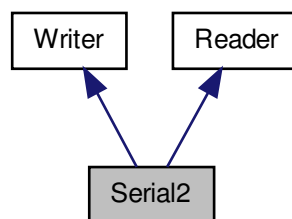
- [USART1.h](#)
- [USART1.cpp](#)

9.8 Serial2 Class Reference

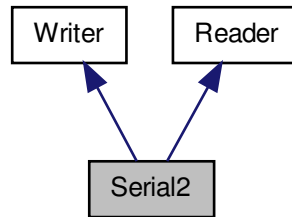
Provides a high-end interface to serial communications using USART2.

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

Inheritance diagram for Serial2:



Collaboration diagram for Serial2:



Public 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\(\)](#).

Additional Inherited Members

9.8.1 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 USART↵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.

9.8.2 Member Function Documentation

9.8.2.1 `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](#).

9.8.2.2 `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](#).

9.8.2.3 `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](#).

9.8.2.4 `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.

9.8.2.5 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.

9.8.2.6 size_t Serial2::write (char c) [virtual]

Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).

- `c` the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.8.2.7 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](#).

9.8.2.8 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](#).

9.8.2.9 `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:

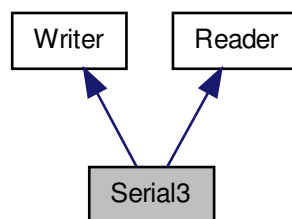
- [USART2.h](#)
- [USART2.cpp](#)

9.9 Serial3 Class Reference

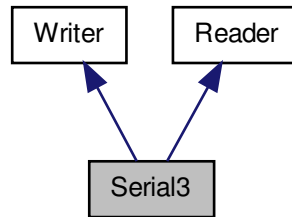
Provides a high-end interface to serial communications using USART3.

```
#include <USART3.h>
```

Inheritance diagram for Serial3:



Collaboration diagram for Serial3:



Public 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\(\)](#).

Additional Inherited Members

9.9.1 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 USART↵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.

9.9.2 Member Function Documentation

9.9.2.1 `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](#).

9.9.2.2 `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](#).

9.9.2.3 `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](#).

9.9.2.4 `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.

9.9.2.5 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.

9.9.2.6 size_t Serial3::write (char c) [virtual]

Write a single character to the output stream. This implements the pure virtual function [Writer::write\(char c \)](#).

- `c` the character to be written.

Returns

the number of bytes written.

Implements [Writer](#).

9.9.2.7 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](#).

9.9.2.8 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](#).

9.9.2.9 `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:

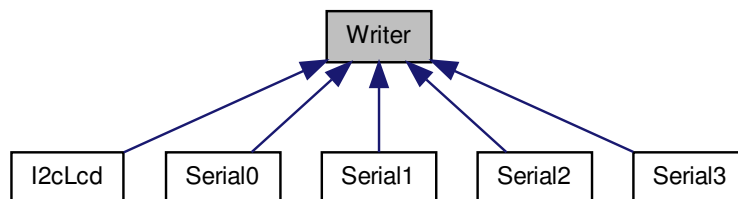
- `USART3.h`
- `USART3.cpp`

9.10 Writer Class Reference

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

```
#include <Writer.h>
```

Inheritance diagram for `Writer`:



Public Types

- enum `IntegerOutputBase` { `kBin` = 2, `kOct` = 8, `kDec` = 10, `kHex` = 16 }

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

Public Member Functions

- virtual size_t [write](#) (char c)=0
Pure virtual function that writes a single character to the output stream.
- virtual size_t [write](#) (const char *str)=0
Pure virtual function that writes a null-terminated string to the output stream.
- virtual size_t [write](#) (const char *buffer, size_t size)=0
Pure virtual function that writes a given number of characters from a buffer to the output stream.
- virtual size_t [write](#) (const uint8_t *buffer, size_t size)=0
Pure virtual function that writes a given number of bytes from a buffer to the output stream.
- virtual void [flush](#) ()=0
Pure virtual function to flush the output stream. When this function returns, all previously written data will have been transmitted through the underlying output stream.
- size_t [print](#) (const char *str, bool addLn=false)
Print a null-terminated string to the output stream, with or without adding a new line character at the end.
- size_t [print](#) (const uint8_t *buf, size_t size, bool addLn=false)
Print a number of bytes to the output stream, with or without adding a new line character at the end.
- size_t [print](#) (char c, bool addLn=false)
Print a single character to the output stream, with or without adding a new line character at the end.
- size_t [print](#) (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.*

9.10.1 Detailed Description

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

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

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

9.10.2 Member Enumeration Documentation

9.10.2.1 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)

9.10.3 Member Function Documentation

9.10.3.1 `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 ('`\n`') is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.2 `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 ('`\n`') is added at the end of the output; the default is false.

Returns

the number of bytes sent to the output stream.

9.10.3.3 `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.

9.10.3.4 `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.

9.10.3.5 `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

the number of bytes sent to the output stream.

9.10.3.6 `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.

9.10.3.7 `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.

9.10.3.8 `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

the number of bytes sent to the output stream.

9.10.3.9 `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.

9.10.3.10 `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.

9.10.3.11 `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.

9.10.3.12 `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

the number of bytes sent to the output stream.

9.10.3.13 `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.

9.10.3.14 `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.

9.10.3.15 `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

the number of bytes sent to the output stream.

9.10.3.16 `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.

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

Returns

the number of bytes sent to the output stream.

9.10.3.18 `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](#).

9.10.3.19 `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](#).

9.10.3.20 `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](#).

9.10.3.21 `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](#)

Chapter 10

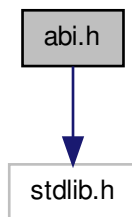
File Documentation

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`.

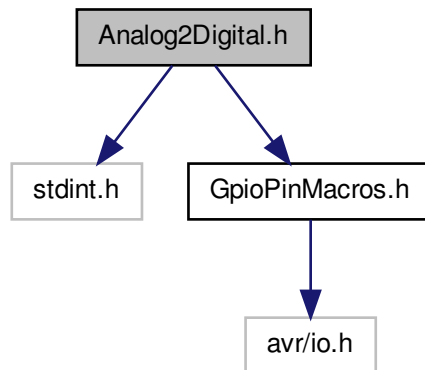
10.2 Analog2Digital.h File Reference

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

- enum `A2DVoltageReference` { `kA2dReferenceAREF`, `kA2dReferenceAVCC`, `kA2dReference11V`, `kA2dReference256V` }
Constants representing voltage references.

Functions

- int `readA2D` (int8_t channel)
Read an analog voltage value.
- uint16_t `readGpioPinAnalogV` (const `GpioPinVariable` &pinVar)
Read the analog value of the pin.
- void `initA2D` (uint8_t ref=`kA2dReferenceAVCC`)
Initialize the analog-to-digital system.
- void `turnOffA2D` ()
Turn off the analog-to-digital system.

- void [setA2DVoltageReference](#) ([A2DVoltageReference](#) ref)
Set the voltage reference for the analog-to-digital system.
- void [setA2DVoltageReferenceAREF](#) ()
Set the voltage reference for the analog-to-digital system to AREF.
- void [setA2DVoltageReferenceAVCC](#) ()
Set the voltage reference for the analog-to-digital system to AREF.
- void [setA2DVoltageReference11V](#) ()
Set the voltage reference for the analog-to-digital system to AREF.
- void [setA2DVoltageReference256V](#) ()
Set the voltage reference for the analog-to-digital system to AREF.

10.2.1 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 you 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.

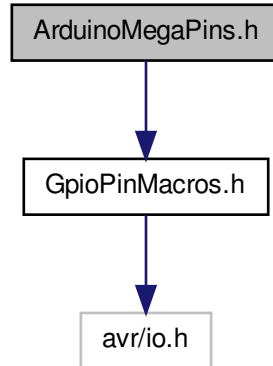
This is an inline synonym for `setA2DVoltageReference(kA2dReferenceAVCC)`

10.3 ArduinoMegaPins.h File Reference

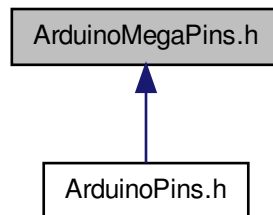
This file defines the standard Arduino Uno pin name macros. It may be included directly by user code, although more commonly user code includes the file [ArduinoPins.h](#), which in turn includes this file (when compiling for Arduino Uno targets).

```
#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`.

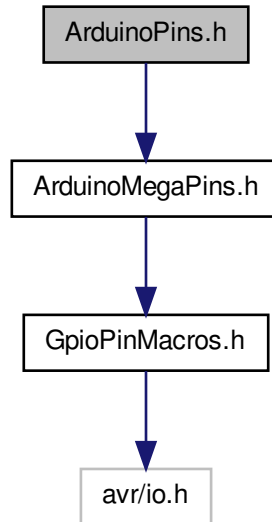
Additionally, the I2C SDA and SCL pins are also defined as `pSDA` and `pSCL` (these are synonyms for `pPin20` and `pPin21`, respectively).

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).

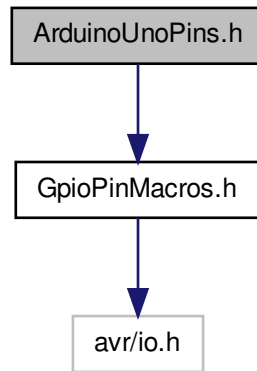
The standard Arduino analog pins will be defined in the form `pPinAxx` (where `xx` = 00 through 07 for Arduino Uno, and `xx` = 00 through 15 for Arduino Mega).

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`.

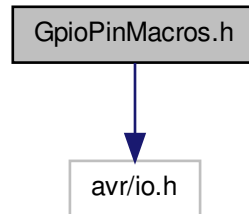
Additionally, the I2C SDA and SCL pins are also defined as `pSDA` and `pSCL` (these are synonyms for `pPinA04` and `pPinA05`, respectively).

10.6 GpioPinMacros.h File Reference

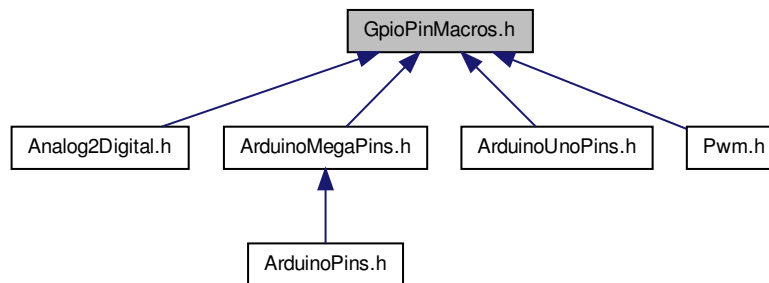
This file contains the primary macros for naming and manipulating GPIO pin names.

```
#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 [setGpioPinModeOutput](#)(pinName)

- Set the mode of the GPIO pin to output (i.e., set the corresponding DDRn bit).*

 - #define `setGpioPinModeInput`(pinName)

Set the mode of the GPIO pin to input (i.e., clear the corresponding DDRn and PORTn bits).
- #define `setGpioPinModeInputPullup`(pinName)

Set the mode of the GPIO pin to input with pullup (i.e., clear the corresponding DDRn bit and set the PORTn bit).
- #define `readGpioPinDigital`(pinName)

Read the value of the GPIO pin (i.e., return the value of corresponding the PINn bit).
- #define `writeGpioPinDigital`(pinName, val)

Write a value the GPIO pin (i.e., set or clear the corresponding the PORTn bit).
- #define `setGpioPinHigh`(pinName)

Write a 1 to the GPIO pin (i.e., set the corresponding the PORTn bit).
- #define `setGpioPinLow`(pinName)

Write a 0 the GPIO pin (i.e., clear the corresponding the PORTn bit).
- #define `getGpioDDR`(pinName)

Get the DDRn corresponding to this GPIO pin.
- #define `getGpioPORT`(pinName)

Get the PORTn corresponding to this GPIO pin.
- #define `getGpioPIN`(pinName)

Get the bit number corresponding to this GPIO pin.
- #define `getGpioMASK`(pinName)

Get the bit mask corresponding to this GPIO pin.
- #define `getGpioADC`(pinName)

Get the ADC channel corresponding to this GPIO pin, assuming it is an ADC capable GPIO pin.
- #define `getGpioOCR`(pinName)

Get the OCR register corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.
- #define `getGpioCOM`(pinName)

Get the COM bit name corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.
- #define `getGpioTCCR`(pinName)

Get the TCCR register corresponding to this GPIO pin, assuming it is a PWM capable GPIO pin.
- #define `makeGpioVarFromGpioPin`(pinName)

Create a GPIO pin variable of type `GpioPinVariable` from a GPIO pin macro.
- #define `makeGpioVarFromGpioPinAnalog`(pinName)

Create a GPIO pin variable of type `GpioPinVariable` that can be used for analog-to-digital reading from a GPIO pin macro.
- #define `makeGpioVarFromGpioPinPwm`(pinName)

Create a GPIO pin variable of type `GpioPinVariable` that can be used for PWM from a GPIO pin macro.

Enumerations

- enum { `kDigitalLow` = 0, `kDigitalHigh` = 1 }

Constants for digital values representing LOW and HIGH.

Functions

- void [setGpioPinModeOutputV](#) (const [GpioPinVariable](#) &pinVar)
Set the mode of the GPIO pin to output (i.e., set the corresponding DDRn bit).
- void [setGpioPinModeInputV](#) (const [GpioPinVariable](#) &pinVar)
Set the mode of the GPIO pin to input (i.e., clear the corresponding DDRn and PORTn bits).
- void [setGpioPinModeInputPullupV](#) (const [GpioPinVariable](#) &pinVar)
Set the mode of the GPIO pin to input with pullup (i.e., clear the corresponding DDRn bit and set the PORTn bit).
- bool [readGpioPinDigitalV](#) (const [GpioPinVariable](#) &pinVar)
Read the value of the GPIO pin (i.e., return the value of corresponding the PINn bit).
- void [writeGpioPinDigitalV](#) (const [GpioPinVariable](#) &pinVar, bool value)
Write a value the GPIO pin (i.e., set or clear the corresponding the PORTn bit).
- void [setGpioPinHighV](#) (const [GpioPinVariable](#) &pinVar)
Write a 1 to the GPIO pin (i.e., set the corresponding the PORTn bit).
- void [setGpioPinLowV](#) (const [GpioPinVariable](#) &pinVar)
Write a 0 to the GPIO pin (i.e., clear the corresponding the PORTn bit).

10.6.1 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 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](#).

10.6.2.13 `#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.

10.6.2.14 `#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.

10.6.2.15 `#define readGpioPinDigital(pinName)`

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

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

Returns

0 or 1

10.6.2.16 `#define setGpioPinHigh(pinName)`

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

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.6.2.17 `#define setGpioPinLow(pinName)`

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

- `pinName` a GPIO pin name macro generated by either [GpioPin\(\)](#), [GpioPinAnalog\(\)](#), or [GpioPinPwm\(\)](#).

10.6.2.18 `#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\(\)](#).

10.6.2.19 `#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\(\)](#).

10.6.2.20 `#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\(\)](#).

10.6.2.21 `#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 readGpioPinDigitalV(const GpioPinVariable & pinVar) [inline]`

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

- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).

Returns

0 or 1

10.6.4.2 `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](#).

10.6.4.3 `void setGpioPinLowV (const GpioPinVariable & pinVar) [inline]`

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

- `pinVar` aa GPIO pin variable of type [GpioPinVariable](#).

10.6.4.4 `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](#).

10.6.4.5 `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](#).

10.6.4.6 `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](#).

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

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

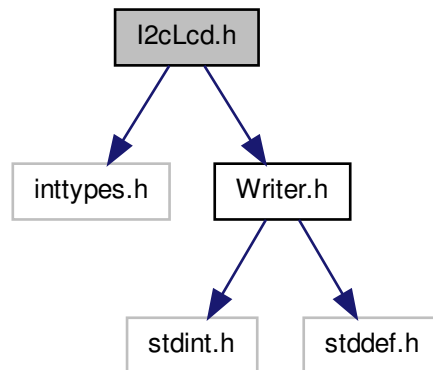
- `pinVar` a GPIO pin variable of type [GpioPinVariable](#).
- `val` the value to be written: 0 means to clear the GPIO pin; any other value means to set it.

10.7 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 associated 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](#).

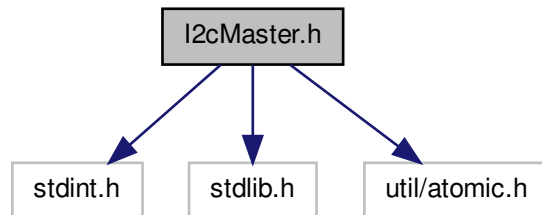
To use these features, include [I2cLcd.h](#) in your source code and link against I2cLcd.cpp and I2cMaster.cpp.

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:
```



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

- enum [I2cMaster::I2cBusSpeed](#) { [I2cMaster::kI2cBusSlow](#), [I2cMaster::kI2cBusFast](#) }

This enum lists I2C bus speed configurations.

- enum [I2cMaster::I2cStatusCodes](#) { [I2cMaster::kI2cCompletedOk](#) = 0x00, [I2cMaster::kI2cError](#) = 0x01, [I2cMaster::kI2cNotStarted](#) = 0x02, [I2cMaster::kI2cInProgress](#) = 0x04 }

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

- enum [I2cMaster::I2cSendErrorCodes](#) { [I2cMaster::kI2cNoError](#) = 0, [I2cMaster::kI2cErrTxBufferFull](#) = 1, [I2cMaster::kI2cErrMsgTooLong](#) = 2, [I2cMaster::kI2cErrNullStatusPtr](#) = 3, [I2cMaster::kI2cErrWriteWithoutData](#) = 4, [I2cMaster::kI2cErrReadWithoutStorage](#) = 5 }

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

- enum [I2cMaster::I2cPullups](#) { [I2cMaster::kPullupsOff](#), [I2cMaster::kPullupsOn](#) }

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

Functions

- void [I2cMaster::start](#) (uint8_t speed=[kI2cBusFast](#))

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

- void [I2cMaster::stop](#) ()

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

- void **I2cMaster::pullups** (uint8_t set=kPullupsOn)
Sets the state of the internal pullups that are part of the TWI hardware.
- bool **I2cMaster::busy** ()
Reports whether the TWI hardware is busy communicating (either transmitting or receiving).
- uint8_t **I2cMaster::writeAsync** (uint8_t address, uint8_t registerAddress, volatile uint8_t *status)
Transmit a single register address (a one-byte message) asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- uint8_t **I2cMaster::writeAsync** (uint8_t address, uint8_t registerAddress, uint8_t data, volatile uint8_t *status)
Transmit a single register address and corresponding single byte of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- uint8_t **I2cMaster::writeAsync** (uint8_t address, uint8_t registerAddress, const char *data, volatile uint8_t *status)
Transmit a single register address and corresponding null-terminated string of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- uint8_t **I2cMaster::writeAsync** (uint8_t address, uint8_t registerAddress, uint8_t *data, uint8_t numberBytes, volatile uint8_t *status)
Transmit a single register address and corresponding buffer of data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function).
- uint8_t **I2cMaster::readAsync** (uint8_t address, uint8_t numberBytes, volatile uint8_t *destination, volatile uint8_t *bytesRead, volatile uint8_t *status)
Request to read data from a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports kI2cCompletedOk, the requested data can be read from the receive buffer.
- uint8_t **I2cMaster::readAsync** (uint8_t address, uint8_t registerAddress, uint8_t numberBytes, volatile uint8_t *destination, volatile uint8_t *bytesRead, volatile uint8_t *status)
Request to read data from a specific register on a device and receive that data asynchronously. This function queues the message and returns immediately. Eventual status of the transmitted message can be monitored via the designated status variable (passed as a pointer to this function). When the status variable reports kI2cCompletedOk, the requested data can be read from the receive buffer.
- int **I2cMaster::writeSync** (uint8_t address, uint8_t registerAddress)
Transmit a single register address (a one-byte message) synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int **I2cMaster::writeSync** (uint8_t address, uint8_t registerAddress, uint8_t data)
Transmit a single register address and corresponding single byte of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int **I2cMaster::writeSync** (uint8_t address, uint8_t registerAddress, const char *data)
Transmit a single register address and corresponding null-terminated string of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int **I2cMaster::writeSync** (uint8_t address, uint8_t registerAddress, uint8_t *data, uint8_t numberBytes)
Transmit a single register address and corresponding buffer of data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int **I2cMaster::readSync** (uint8_t address, uint8_t numberBytes, uint8_t *destination)
Request to read data from a device and receive that data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).
- int **I2cMaster::readSync** (uint8_t address, uint8_t registerAddress, uint8_t numberBytes, uint8_t *destination)
Request to read data from a specific register on a device and receive that data synchronously. This function blocks until the communications exchange is complete or encounters an error. Error codes are returned (0 means no error).

10.8.1 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_TX_MSG_NBR` to specify the maximum number of transmit messages to hold in the buffer. You need to make these define these macros prior to including the file [I2cMaster.h](#), each time it is included. So you should define these using a compiler option (e.g., `-DI2C_MASTER_MAX_TX_MSG_LEN=32 -DI2C_MASTER_MAX_TX_MSG_NBR=5`) to ensure they are consistently defined throughout your project.

This interface assumes your application will operator in I2C Master mode as defined in the I2C protocol. If you wish your application to operate in I2C Slave mode, then instead include [I2cSlave.h](#) and link against `I2cSlave.cpp`.

Note

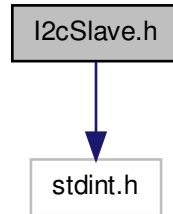
Only one of `I2cMaster.cpp` and `I2cSlave.cpp` can be linking into your application. These two files install different, incompatible versions of the TWI interrupt function. AVRTools does not support building an application that functions both as a Master and as a Slave under the I2C protocol. This limitation allows the corresponding TWI interrupt functions to be significantly leaner and faster.

10.9 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

- enum [I2cSlave::I2cBusSpeed](#) { [I2cSlave::kI2cBusSlow](#), [I2cSlave::kI2cBusFast](#) }

This enum lists I2C bus speed configurations.

- enum [I2cSlave::I2cStatusCodes](#) {
[I2cSlave::kI2cCompletedOk](#) = 0x00, [I2cSlave::kI2cError](#) = 0x01, [I2cSlave::kI2cTxPartial](#) = 0x02, [I2cSlave::kI2cRxOverflow](#) = 0x04,
[I2cSlave::kI2cInProgress](#) = 0x06 }

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

- enum [I2cSlave::I2cPullups](#) { [I2cSlave::kPullupsOff](#), [I2cSlave::kPullupsOn](#) }

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

Functions

- [uint8_t I2cSlave::processI2cMessage](#) (uint8_t *buffer, uint8_t len)

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

- void [I2cSlave::start](#) (uint8_t ownAddress, uint8_t speed=kI2cBusFast, bool answerGeneralCall=false)

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

- void [I2cSlave::stop](#) ()

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

- void [I2cSlave::pullups](#) (uint8_t set=kPullupsOn)

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

- bool [I2cSlave::busy](#) ()

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

10.9.1 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 receiving and sending data and operate using interrupts associated with the TWI hardware. This means data from the Master is received asynchronously and when reception is complete, a user-supplied function is called. That function has the option of placing data in a buffer to be transmitted asynchronously back to the Master.

The Slave buffer is a simple array. The size of the Slave buffer can be set at compile time via the macro constant `I2C_SLAVE_BUFFER_SIZE`. The default size of the Slave buffer is 32 bytes. You can change the default by defining the macro `I2C_SLAVE_BUFFER_SIZE` prior to including the file [I2cSlave.h](#), each time it is included. So you should define it using a compiler option (e.g., `-DI2C_SLAVE_BUFFER_SIZE=64`) to ensure it is consistently defined throughout your project.

This interface assumes your application will operator in I2C Slave mode as defined in the I2C protocol. If you wish your application to operate in I2C Master mode, then instead include [I2cMaster.h](#) and link against I2cMaster.cpp.

Note

Only one of I2cMaster.cpp and I2cSlave.cpp can be linking into your application. These two files install different, incompatible versions of the TWI interrupt function. AVRTools does not support building an application that functions both as a Master and as a Slave under the I2C protocol. This limitation allows the corresponding TWI interrupt functions to be significantly leaner and faster.

10.10 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 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.

10.11.1 Detailed Description

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

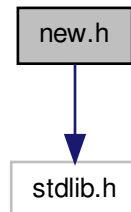
To use these functions, include [MemUtils.h](#) in your source code and link against `MemUtils.cpp`.

These functions are wrapped in namespace [MemUtils](#) to avoid namespace collisions.

10.12 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:
```



10.12.1 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

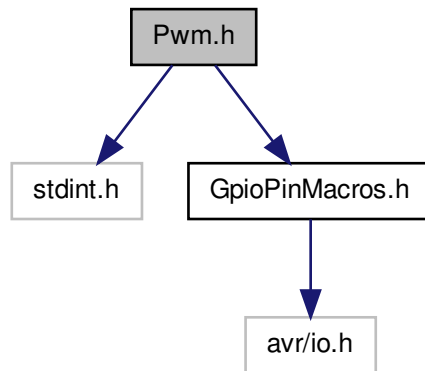
The AVRTools library does not itself make any use of heap storage or the `new` or `delete` operators.

10.13 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.

10.13.1 Detailed Description

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

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

Before you use the `writePinPwm()` function, you must first initialize the appropriate timers using the appropriate `initPwmTimerN()` function.

The association between PWN pins and timers is as follows:

For Arduino Uno (ATmega328)

Arduino Uno pin	ATmega328 pin	Timer
3	PD3	timer2
5	PD5	timer0
6	PD6	timer0
9	PB1	timer1
10	PB2	timer1
11	PB3	timer2

For Arduino Mega (ATmega2560)

Arduino Mega pin	ATmega2560 pin	Timer
2	PE4	timer3
3	PE5	timer3
4	PG5	timer0
5	PE3	timer3
6	PH3	timer4
7	PH4	timer4
8	PH5	timer4
9	PH6	timer2
10	PB4	timer2
11	PB5	timer1
12	PB6	timer1
13	PB7	timer0
44	PL5	timer5
45	PL4	timer5
46	PL3	timer5

Note

Timer0 is also used by the system clock. *Do not initialize or clear timer0* if you are also using the system clock function from `SystemClock.h`. If you are using the system clock function, you can use timer0-based PWM functions *without* having to call `initPwmTimer0()`.

10.13.2 Macro Definition Documentation

10.13.2.1 `#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.

10.13.3 Function Documentation

10.13.3.1 `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)`.

10.13.3.2 `void clearTimer1 ()`

Clear timer1.

This function clears timer1, turning off the PWM functionality.

10.13.3.3 `void clearTimer2 ()`

Clear timer2.

This function clears timer2, turning off the PWM functionality.

10.13.3.4 void clearTimer3 ()

Clear timer3.

This function clears timer3, turning off the PWM functionality.

Note

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

10.13.3.5 void clearTimer4 ()

Clear timer4.

This function clears timer4, turning off the PWM functionality.

Note

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

10.13.3.6 void clearTimer5 ()

Clear timer5.

This function clears timer5, turning off the PWM functionality.

Note

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

10.13.3.7 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)`.

10.13.3.8 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)

10.13.3.9 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)

10.13.3.10 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).

10.13.3.11 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).

10.13.3.12 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).

10.13.3.13 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.

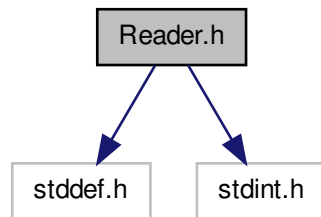
10.14 Reader.h File Reference

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.

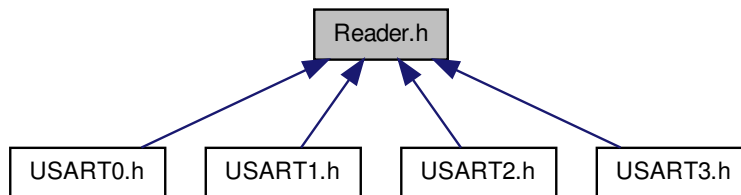
```
#include <stddef.h>
```

```
#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).

10.14.1 Detailed Description

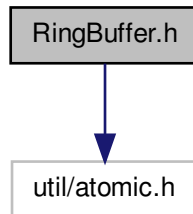
This file provides a generic interface to incoming data streams of any kind. It is designed around how serial streams are generally used, but can be used with any system that provides a sequential input of bytes that can be interpreted as strings and/or numbers.

10.15 RingBuffer.h File Reference

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

```
#include <util/atomic.h>
```

Include dependency graph for RingBuffer.h:



Classes

- class [RingBuffer](#)

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

10.15.1 Detailed Description

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

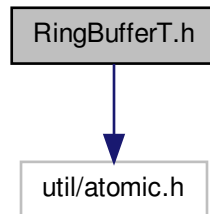
Ring buffers are particularly useful for memory constrained microcontrollers such as the ATmega328 and ATmega2650.

10.16 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.

10.16.1 Detailed Description

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

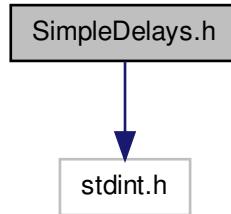
Ring buffers are versatile storage structures. This file provides a template-based ring buffer implementation that can store different kinds of objects in buffers of whatever size is needed.

10.17 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, the smallest possible delay is just under 6 quarter microseconds (~1.5 microseconds). Delays of 7 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.

10.17.1 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 all assume a 16 MHz clock cycle.

10.17.2 Function Documentation

10.17.2.1 void [delayQuartersOfMicroSeconds](#) (uint16_t *nbrOfQuartersOfMicroSeconds*)

Delay a given number of quarter microseconds. Due to function call overhead, the smallest possible delay is just under 6 quarter microseconds (~1.5 microseconds). Delays of 7 quarter microseconds or greater are reasonably accurate.

Delays of less than 7 quarter microseconds produce a delay of just under 6 quarter microseconds (~1.5 microseconds). The maximum delay is 65535 quarter microseconds (equal to 16,383.75 microseconds, or about 16.4 milliseconds).

- `nbrOfQuartersOfMicroSeconds` the number of quarter microseconds to delay. Arguments less than 7 quarter microseconds all produce delays of just under 6 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 assumes a 16 MHz clock rate.

For precision, this function is implemented directly in assembler.

10.17.2.2 void delayTenthsOfSeconds (uint8_t *nrOfTenthsOfSeconds*)

Delay a given number of tenths of a seconds. Despite function call overhead, this function is accurate within a few hundreds of microseconds.

- `nrOfTenthsOfSeconds` 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 assumes a 16 MHz clock rate.

For precision, this function is implemented directly in assembler.

10.17.2.3 void delayWholeMilliseconds (uint8_t *nrOfMilliSeconds*)

Delay a given number of milliseconds. Despite function call overhead, this function is accurate within a few hundreds of microseconds.

- `nrOfMilliSeconds` 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 assumes a 16 MHz clock rate.

For precision, this function is implemented directly in assembler.

10.18 SystemClock.h File Reference

Include this file to use the functions that instantiate and access a system clock that counts elapsed milliseconds.

Functions

- void `initSystemClock()`
This function initializes a system clock that tracks elapsed milliseconds.
- void `delayMicroseconds` (unsigned int *us*)
Delay a certain number of microseconds.
- void `delayMilliseconds` (unsigned long *ms*)
Delay a certain number of milliseconds.
- void `delay` (unsigned long *ms*)
Delay a certain number of milliseconds.
- unsigned long `micros` ()
Return the number of elapsed microseconds since the system clock was turned on.
- unsigned long `millis` ()
Return the number of elapsed milliseconds since the system clock was turned on.

10.18.1 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`.

10.18.2 Function Documentation

10.18.2.1 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.

10.18.2.2 void delayMicroseconds (unsigned int *us*)

Delay a certain number of microseconds.

- *us* the number of microseconds to delay.

10.18.2.3 void delayMilliseconds (unsigned long *ms*)

Delay a certain number of milliseconds.

- *m* the number of milliseconds to delay.

10.18.2.4 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.

10.18.2.5 unsigned long micros ()

Return the number of elapsed microseconds since the system clock was turned on.

The microsecond count will overflow back to zero in approximately 70 minutes.

Returns

the number of elapsed microseconds.

10.18.2.6 unsigned long millis ()

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

The millisecond count will overflow back to zero in approximately 50 days.

Returns

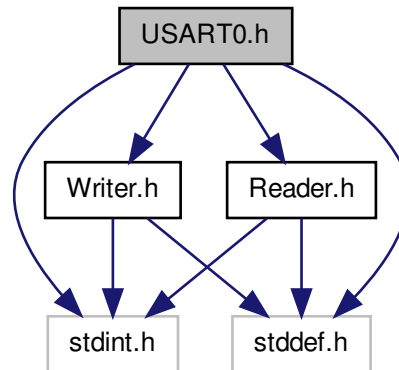
the number of elapsed milliseconds.

10.19 USART0.h File Reference

This file provides functions that offer high-level interfaces to USART0 hardware, which is available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560).

```
#include "Writer.h"
#include "Reader.h"
#include <stdint.h>
#include <stddef.h>
```

Include dependency graph for USART0.h:



Classes

- class [Serial0](#)

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

Namespaces

- [USART0](#)

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

Enumerations

- enum `UsartSerialConfiguration` {
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2` }

This enum lists serial configuration in terms of data bits, parity, and stop bits.

Functions

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

10.19.1 Detailed Description

This file provides functions that offer high-level interfaces to USART0 hardware, which is available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560).

These interfaces are buffered for both input and output and operate using interrupts associated with USART0. This means the transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART0 hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit and receive buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The receive buffer, however, will overwrite if it gets full. You must clear the receive buffer by reading it regularly when receiving significant amounts of data.

The sizes of the transmit and receive buffers can be set at compile time via macro constants. The default sizes are 32 bytes for the receive buffer and 64 bytes for the transmit buffer. To change these, define the macros `USART0_RX_BUFFER_SIZE` (for the receive buffer) and `USART0_TX_BUFFER_SIZE` (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file `USART0.cpp`.

Two interfaces are provided. `USART0` is a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontrollers. However, `USART0` is limited to transmitting and receiving byte and character streams.

`Serial0` is the most advanced and capable interface to the USART0 hardware. `Serial0` provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously.

To use these functions, include `USART0.h` in your source code and link against `USART0.cpp`.

Note

Linking against `USART0.cpp` installs interrupt functions for transmit and receive on USART0 (interrupts `USART0_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`.

10.19.2 Enumeration Type Documentation

10.19.2.1 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_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits
kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits

kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits
kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit

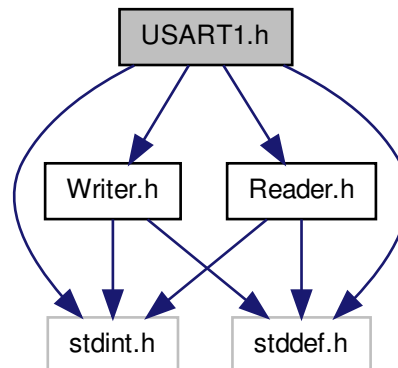
kSerial_601 6 data bits, odd parity, 1 stop bit
kSerial_701 7 data bits, odd parity, 1 stop bit
kSerial_801 8 data bits, odd parity, 1 stop bit
kSerial_502 5 data bits, odd parity, 2 stop bits
kSerial_602 6 data bits, odd parity, 2 stop bits
kSerial_702 7 data bits, odd parity, 2 stop bits
kSerial_802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

10.20 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

- enum `UsartSerialConfiguration` {
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2` }

This enum lists serial configuration in terms of data bits, parity, and stop bits.

Functions

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

10.20.1 Detailed Description

This file provides functions that offer high-level interfaces to USART1 hardware, which is available on Arduino Mega (ATMega2560).

These interfaces are buffered for both input and output and operate using interrupts associated with USART1. This means the transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART1 hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit and receive buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The receive buffer, however, will overwrite if it gets full. You must clear the receive buffer by reading it regularly when receiving significant amounts of data.

The sizes of the transmit and receive buffers can be set at compile time via macro constants. The default sizes are 32 bytes for the receive buffer and 64 bytes for the transmit buffer. To change these, define the macros `USART1_RX_BUFFER_SIZE` (for the receive buffer) and `USART1_TX_BUFFER_SIZE` (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file `USART1.cpp`.

Two interfaces are provided. `USART1` is a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontrollers. However, `USART1` is limited to transmitting and receiving byte and character streams.

`Serial1` is the most advanced and capable interface to the USART1 hardware. `Serial1` provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously.

To use these functions, include `USART1.h` in your source code and link against `USART1.cpp`.

Note

Linking against `USART1.cpp` installs interrupt functions for transmit and receive on USART1 (interrupts `USART1_UDRE` and `USART1_RX` on Arduino Mega/ATmega2560). You cannot use the minimal interface to USART1 (from `USARTMinimal.h`) if you link against `USART1.cpp`. In particular, do *not* call `initUSART1()` or `clearUSART1()` if you link against `USART1.cpp`.

10.20.2 Enumeration Type Documentation

10.20.2.1 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_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits
kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits

kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits
kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit

kSerial_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

10.21 USART2.h File Reference

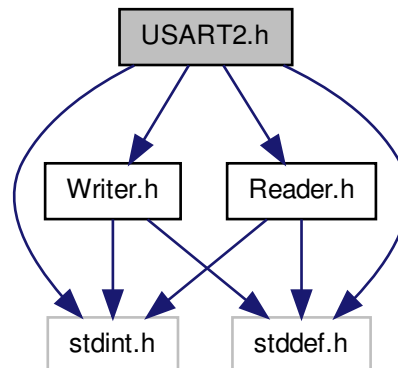
This file provides functions that offer high-level interfaces to USART2 hardware, which is available on Arduino Mega (ATMega2560).

```

#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

- enum `UsartSerialConfiguration` {
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2` }

This enum lists serial configuration in terms of data bits, parity, and stop bits.

Functions

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

10.21.1 Detailed Description

This file provides functions that offer high-level interfaces to USART2 hardware, which is available on Arduino Mega (ATMega2560).

These interfaces are buffered for both input and output and operate using interrupts associated with USART2. This means the transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART2 hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit and receive buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The receive buffer, however, will overwrite if it gets full. You must clear the receive buffer by reading it regularly when receiving significant amounts of data.

The sizes of the transmit and receive buffers can be set at compile time via macro constants. The default sizes are 32 bytes for the receive buffer and 64 bytes for the transmit buffer. To change these, define the macros `USART2_RX_BUFFER_SIZE` (for the receive buffer) and `USART2_TX_BUFFER_SIZE` (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file `USART2.cpp`.

Two interfaces are provided. `USART2` is a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontrollers. However, `USART2` is limited to transmitting and receiving byte and character streams.

`Serial2` is the most advanced and capable interface to the USART2 hardware. `Serial2` provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously.

To use these functions, include `USART2.h` in your source code and link against `USART2.cpp`.

Note

Linking against `USART2.cpp` installs interrupt functions for transmit and receive on USART2 (interrupts `USART2_UDRE` and `USART2_RX` on Arduino Mega/ATmega2560). You cannot use the minimal interface to USART2 (from `USARTMinimal.h`) if you link against `USART2.cpp`. In particular, do *not* call `initUSART2()` or `clearUSART2()` if you link against `USART2.cpp`.

10.21.2 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_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits
kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits

kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits
kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit

kSerial_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 USART3.h File Reference

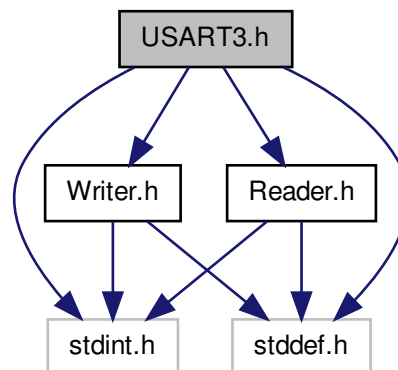
This file provides functions that offer high-level interfaces to USART3 hardware, which is available on Arduino Mega (ATMega2560).

```

#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

- enum `UsartSerialConfiguration` {
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2`,
`kSerial_5N1`, `kSerial_6N1`, `kSerial_7N1`, `kSerial_8N1`,
`kSerial_5N2`, `kSerial_6N2`, `kSerial_7N2`, `kSerial_8N2`,
`kSerial_5E1`, `kSerial_6E1`, `kSerial_7E1`, `kSerial_8E1`,
`kSerial_5E2`, `kSerial_6E2`, `kSerial_7E2`, `kSerial_8E2`,
`kSerial_5O1`, `kSerial_6O1`, `kSerial_7O1`, `kSerial_8O1`,
`kSerial_5O2`, `kSerial_6O2`, `kSerial_7O2`, `kSerial_8O2` }

This enum lists serial configuration in terms of data bits, parity, and stop bits.

Functions

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

10.22.1 Detailed Description

This file provides functions that offer high-level interfaces to USART3 hardware, which is available on Arduino Mega (ATMega2560).

These interfaces are buffered for both input and output and operate using interrupts associated with USART3. This means the transmit functions return immediately after queuing data in the output buffer for transmission and the transmission happens asynchronously, using dedicated USART3 hardware. Similarly, data is received asynchronously and placed into the input buffer.

The transmit and receive buffers are both ring buffers. If you try to queue more data than the transmit buffer can hold, the write functions will block until there is room in the buffer (as a result of data being transmitted). The receive buffer, however, will overwrite if it gets full. You must clear the receive buffer by reading it regularly when receiving significant amounts of data.

The sizes of the transmit and receive buffers can be set at compile time via macro constants. The default sizes are 32 bytes for the receive buffer and 64 bytes for the transmit buffer. To change these, define the macros `USART3_RX_BUFFER_SIZE` (for the receive buffer) and `USART3_TX_BUFFER_SIZE` (for the transmit buffer) to whatever sizes you need. You need to make these define these macros prior to compiling the file `USART3.cpp`.

Two interfaces are provided. `USART3` is a functional interface that makes use of the buffering and asynchronous transmit and receive capabilities of the microcontrollers. However, `USART3` is limited to transmitting and receiving byte and character streams.

`Serial3` is the most advanced and capable interface to the USART3 hardware. `Serial3` provides a object-oriented interface that includes the ability to read and write numbers of various types and in various formats, all asynchronously.

To use these functions, include `USART3.h` in your source code and link against `USART3.cpp`.

Note

Linking against `USART3.cpp` installs interrupt functions for transmit and receive on USART3 (interrupts `USART3_UDRE` and `USART3_RX` on Arduino Mega/ATmega2560). You cannot use the minimal interface to USART3 (from `USARTMinimal.h`) if you link against `USART3.cpp`. In particular, do *not* call `initUSART3()` or `clearUSART3()` if you link against `USART3.cpp`.

10.22.2 Enumeration Type Documentation

10.22.2.1 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_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits
kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits

kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit
kSerial_6O1 6 data bits, odd parity, 1 stop bit
kSerial_7O1 7 data bits, odd parity, 1 stop bit
kSerial_8O1 8 data bits, odd parity, 1 stop bit
kSerial_5O2 5 data bits, odd parity, 2 stop bits
kSerial_6O2 6 data bits, odd parity, 2 stop bits
kSerial_7O2 7 data bits, odd parity, 2 stop bits
kSerial_8O2 8 data bits, odd parity, 2 stop bits
kSerial_5N1 5 data bits, no parity, 1 stop bit
kSerial_6N1 6 data bits, no parity, 1 stop bit
kSerial_7N1 7 data bits, no parity, 1 stop bit
kSerial_8N1 8 data bits, no parity, 1 stop bit
kSerial_5N2 5 data bits, no parity, 2 stop bits
kSerial_6N2 6 data bits, no parity, 2 stop bits
kSerial_7N2 7 data bits, no parity, 2 stop bits
kSerial_8N2 8 data bits, no parity, 2 stop bits
kSerial_5E1 5 data bits, even parity, 1 stop bit
kSerial_6E1 6 data bits, even parity, 1 stop bit
kSerial_7E1 7 data bits, even parity, 1 stop bit
kSerial_8E1 8 data bits, even parity, 1 stop bit
kSerial_5E2 5 data bits, even parity, 2 stop bits
kSerial_6E2 6 data bits, even parity, 2 stop bits
kSerial_7E2 7 data bits, even parity, 2 stop bits
kSerial_8E2 8 data bits, even parity, 2 stop bits
kSerial_5O1 5 data bits, odd parity, 1 stop bit

kSerial_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

10.23 USARTMinimal.h File Reference

This file provides functions that provide a minimalist interface to the USARTs available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560).

Functions

- void [initUSART0](#) (unsigned long baudRate)
Initialize USART0 for serial receive and transmit.
- void [transmitUSART0](#) (unsigned char data)
Transmit a single byte on USART0.
- void [transmitUSART0](#) (const char *data)
Transmit a null-terminated string on USART0.
- unsigned char [receiveUSART0](#) ()
Receive a byte on USART0.
- void [releaseUSART0](#) ()
Release USART0, making pins 0 and 1 again available for non-USART use.
- 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.
- 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.

- 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.

10.23.1 Detailed Description

This file provides functions that provide a minimalist interface to the USARTs available on the Arduino Uno (ATmega328) and Arduino Mega (ATmega2560).

These functions are minimalist in the following sense:

- They only send single bytes or zero-terminated character strings.
- They only receive single characters.
- They do not use the USART-related interrupts.
- They determine when the USART is ready to send by polling the relevant register bit.
- They determine when the USART has received data by polling the relevant register bit.

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

For a more advanced USART interface, consider using either the `USART0` or `Serial0` interfaces. Both of these are available by including `USART0.h` instead of `USARTMinimal.h`.

10.23.2 Function Documentation

10.23.2.1 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 (ATmega2560 pins PE0, PE1).

Communications are configured for 8 data bits, no parity, and 1 stop bit.

- `baudRate` the baud rate for the communications, usually one of the following values: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200 (although other values below can be specified).

10.23.2.2 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).

10.23.2.3 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).

10.23.2.4 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).

10.23.2.5 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.

10.23.2.6 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).

10.23.2.7 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).

10.23.2.8 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).

10.23.2.9 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\(\)](#).

10.23.2.10 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).

10.23.2.11 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).

10.23.2.12 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).

10.23.2.13 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.

10.23.2.14 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.

10.23.2.15 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).

10.23.2.16 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).

10.23.2.17 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).

10.23.2.18 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).

10.23.2.19 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).

10.23.2.20 void transmitUSART3 (const char * *data*)

Transmit a null-terminated string on [USART3](#).

You must first initialize [USART3](#) by calling [initUSART3\(\)](#).

This function blocks until the USART becomes available and all the bytes can be transmitted.

- `data` the null-terminated string to be transmitted.

Note

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

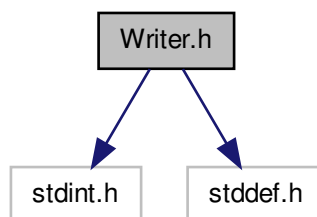
10.24 Writer.h File Reference

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.

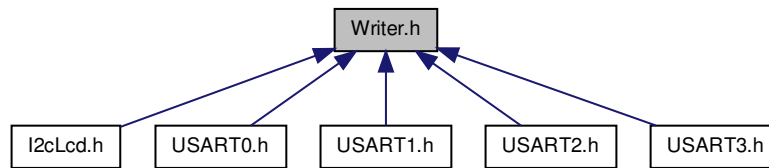
```
#include <stdint.h>
```

```
#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).

10.24.1 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

- A2DVoltageReference
 - Analog2Digital.h, [97](#)
- abi.h, [95](#)
- Analog2Digital.h, [96](#)
 - A2DVoltageReference, [97](#)
 - initA2D, [98](#)
 - kA2dReference11V, [97](#)
 - kA2dReference256V, [97](#)
 - kA2dReferenceAREF, [97](#)
 - kA2dReferenceAVCC, [97](#)
 - readA2D, [98](#)
 - readGpioPinAnalog, [97](#)
 - readGpioPinAnalogV, [98](#)
 - setA2DVoltageReference, [98](#)
 - setA2DVoltageReference11V, [99](#)
 - setA2DVoltageReference256V, [99](#)
 - setA2DVoltageReferenceAREF, [99](#)
 - setA2DVoltageReferenceAVCC, [99](#)
- ArduinoMegaPins.h, [99](#)
- ArduinoPins.h, [101](#)
- ArduinoUnoPins.h, [101](#)
- available
 - Reader, [59](#)
 - Serial0, [71](#)
 - Serial1, [75](#)
 - Serial2, [79](#)
 - Serial3, [83](#)
 - USART0, [38](#)
 - USART1, [41](#)
 - USART2, [44](#)
 - USART3, [47](#)
- busy
 - I2cMaster, [28](#)
 - I2cSlave, [35](#)
- clearTimer0
 - Pwm.h, [121](#)
- clearTimer1
 - Pwm.h, [121](#)
- clearTimer2
 - Pwm.h, [121](#)
- clearTimer3
 - Pwm.h, [121](#)
- clearTimer4
 - Pwm.h, [122](#)
- clearTimer5
 - Pwm.h, [122](#)
- command
 - I2cLcd, [55](#)
- delay
 - SystemClock.h, [130](#)
- delayMicroseconds
 - SystemClock.h, [130](#)
- delayMilliseconds
 - SystemClock.h, [130](#)
- delayQuartersOfMicroSeconds
 - SimpleDelays.h, [128](#)
- delayTenthsOfSeconds
 - SimpleDelays.h, [129](#)
- delayWholeMilliseconds
 - SimpleDelays.h, [129](#)
- discardFromFront
 - RingBufferT, [67](#)
- displayBottomRow
 - I2cLcd, [55](#)
- displayTopRow
 - I2cLcd, [55](#)
- find
 - Reader, [59](#), [60](#)
- findUntil
 - Reader, [60](#)
- flush
 - USART0, [38](#)
 - USART1, [41](#)
 - USART2, [44](#)
 - USART3, [47](#)
- freeRam
 - MemUtils, [37](#)
- freeRamQuickEstimate
 - MemUtils, [37](#)
- getGpioADC
 - GpioPinMacros.h, [105](#)
- getGpioCOM
 - GpioPinMacros.h, [105](#)
- getGpioDDR
 - GpioPinMacros.h, [105](#)
- getGpioMASK
 - GpioPinMacros.h, [106](#)

- getGpioOCR
 - GpioPinMacros.h, 106
- getGpioPIN
 - GpioPinMacros.h, 106
- getGpioPORT
 - GpioPinMacros.h, 106
- getGpioTCCR
 - GpioPinMacros.h, 106
- GpioPin
 - GpioPinMacros.h, 107
- GpioPinAnalog
 - GpioPinMacros.h, 107
- GpioPinMacros.h, 102
 - getGpioADC, 105
 - getGpioCOM, 105
 - getGpioDDR, 105
 - getGpioMASK, 106
 - getGpioOCR, 106
 - getGpioPIN, 106
 - getGpioPORT, 106
 - getGpioTCCR, 106
 - GpioPin, 107
 - GpioPinAnalog, 107
 - GpioPinPwm, 107
 - kDigitalHigh, 109
 - kDigitalLow, 109
 - makeGpioVarFromGpioPin, 107
 - makeGpioVarFromGpioPinAnalog, 107
 - makeGpioVarFromGpioPinPwm, 108
 - readGpioPinDigital, 108
 - readGpioPinDigitalV, 109
 - setGpioPinHigh, 108
 - setGpioPinHighV, 109
 - setGpioPinLow, 108
 - setGpioPinLowV, 110
 - setGpioPinModeInput, 108
 - setGpioPinModeInputPullup, 109
 - setGpioPinModeInputPullupV, 110
 - setGpioPinModeInputV, 110
 - setGpioPinModeOutput, 109
 - setGpioPinModeOutputV, 110
 - writeGpioPinDigital, 109
 - writeGpioPinDigitalV, 110
- GpioPinPwm
 - GpioPinMacros.h, 107
- GpioPinVariable, 51
- I2cBusSpeed
 - I2cMaster, 27
 - I2cSlave, 35
- I2cLcd, 52
 - command, 55
 - displayBottomRow, 55
 - displayTopRow, 55
- init, 55
 - kBacklight_Blue, 55
 - kBacklight_Green, 55
 - kBacklight_Red, 55
 - kBacklight_Teal, 55
 - kBacklight_Violet, 55
 - kBacklight_White, 55
 - kBacklight_Yellow, 55
 - kButton_Down, 55
 - kButton_Left, 55
 - kButton_Right, 55
 - kButton_Select, 55
 - kButton_Up, 55
 - readButtons, 56
 - setBacklight, 56
 - setCursor, 56
 - write, 56, 57
- I2cLcd.h, 111
- I2cMaster, 25
 - busy, 28
 - I2cBusSpeed, 27
 - I2cPullups, 27
 - I2cSendErrorCodes, 27
 - I2cStatusCodes, 28
 - kl2cBusFast, 27
 - kl2cBusSlow, 27
 - kl2cCompletedOk, 28
 - kl2cErrMsgTooLong, 28
 - kl2cErrNullStatusPtr, 28
 - kl2cErrReadWithoutStorage, 28
 - kl2cErrTxBufferFull, 28
 - kl2cErrWriteWithoutData, 28
 - kl2cError, 28
 - kl2cInProgress, 28
 - kl2cNoError, 28
 - kl2cNotStarted, 28
 - kPullupsOff, 27
 - kPullupsOn, 27
 - pullups, 28
 - readAsync, 28, 29
 - readSync, 29, 30
 - start, 30
 - stop, 30
 - writeAsync, 30–32
 - writeSync, 32, 33
- I2cMaster.h, 111
- I2cPullups
 - I2cMaster, 27
 - I2cSlave, 35
- I2cSendErrorCodes
 - I2cMaster, 27
- I2cSlave, 34
 - busy, 35
 - I2cBusSpeed, 35

- I2cPullups, [35](#)
- I2cStatusCodes, [35](#)
- kl2cBusFast, [35](#)
- kl2cBusSlow, [35](#)
- kl2cCompletedOk, [35](#)
- kl2cError, [35](#)
- kl2cInProgress, [35](#)
- kl2cRxOverflow, [35](#)
- kl2cTxPartial, [35](#)
- kPullupsOff, [35](#)
- kPullupsOn, [35](#)
- processI2cMessage, [35](#)
- pullups, [36](#)
- start, [36](#)
- stop, [36](#)
- I2cSlave.h, [114](#)
- I2cStatusCodes
 - I2cMaster, [28](#)
 - I2cSlave, [35](#)
- init
 - I2cLcd, [55](#)
- initA2D
 - Analog2Digital.h, [98](#)
- initPwmTimer0
 - Pwm.h, [122](#)
- initPwmTimer1
 - Pwm.h, [123](#)
- initPwmTimer2
 - Pwm.h, [123](#)
- initPwmTimer3
 - Pwm.h, [123](#)
- initPwmTimer4
 - Pwm.h, [123](#)
- initPwmTimer5
 - Pwm.h, [124](#)
- initSystem
 - InitSystem.h, [116](#)
- InitSystem.h, [116](#)
 - initSystem, [116](#)
- initSystemClock
 - SystemClock.h, [130](#)
- initUSART0
 - USARTMinimal.h, [153](#)
- initUSART1
 - USARTMinimal.h, [153](#)
- initUSART2
 - USARTMinimal.h, [154](#)
- initUSART3
 - USARTMinimal.h, [154](#)
- IntegerOutputBase
 - Writer, [87](#)
- isEmpty
 - RingBuffer, [65](#)
 - RingBufferT, [67](#)
- isFull
 - RingBuffer, [65](#)
 - RingBufferT, [68](#)
- isNotEmpty
 - RingBuffer, [65](#)
 - RingBufferT, [68](#)
- isNotFull
 - RingBuffer, [65](#)
 - RingBufferT, [68](#)
- kA2dReference11V
 - Analog2Digital.h, [97](#)
- kA2dReference256V
 - Analog2Digital.h, [97](#)
- kA2dReferenceAREF
 - Analog2Digital.h, [97](#)
- kA2dReferenceAVCC
 - Analog2Digital.h, [97](#)
- kBacklight_Blue
 - I2cLcd, [55](#)
- kBacklight_Green
 - I2cLcd, [55](#)
- kBacklight_Red
 - I2cLcd, [55](#)
- kBacklight_Teal
 - I2cLcd, [55](#)
- kBacklight_Violet
 - I2cLcd, [55](#)
- kBacklight_White
 - I2cLcd, [55](#)
- kBacklight_Yellow
 - I2cLcd, [55](#)
- kBin
 - Writer, [87](#)
- kButton_Down
 - I2cLcd, [55](#)
- kButton_Left
 - I2cLcd, [55](#)
- kButton_Right
 - I2cLcd, [55](#)
- kButton_Select
 - I2cLcd, [55](#)
- kButton_Up
 - I2cLcd, [55](#)
- kDec
 - Writer, [87](#)
- kDigitalHigh
 - GpioPinMacros.h, [109](#)
- kDigitalLow
 - GpioPinMacros.h, [109](#)
- kHex
 - Writer, [87](#)
- kl2cBusFast
 - I2cMaster, [27](#)

- I2cSlave, [35](#)
- kl2cBusSlow
 - I2cMaster, [27](#)
 - I2cSlave, [35](#)
- kl2cCompletedOk
 - I2cMaster, [28](#)
 - I2cSlave, [35](#)
- kl2cErrMsgTooLong
 - I2cMaster, [28](#)
- kl2cErrNullStatusPtr
 - I2cMaster, [28](#)
- kl2cErrReadWithoutStorage
 - I2cMaster, [28](#)
- kl2cErrTxBufferFull
 - I2cMaster, [28](#)
- kl2cErrWriteWithoutData
 - I2cMaster, [28](#)
- kl2cError
 - I2cMaster, [28](#)
 - I2cSlave, [35](#)
- kl2cInProgress
 - I2cMaster, [28](#)
 - I2cSlave, [35](#)
- kl2cNoError
 - I2cMaster, [28](#)
- kl2cNotStarted
 - I2cMaster, [28](#)
- kl2cRxOverflow
 - I2cSlave, [35](#)
- kl2cTxPartial
 - I2cSlave, [35](#)
- kOct
 - Writer, [87](#)
- kPullupsOff
 - I2cMaster, [27](#)
 - I2cSlave, [35](#)
- kPullupsOn
 - I2cMaster, [27](#)
 - I2cSlave, [35](#)
- kSerial_5E1
 - USART0.h, [135, 136](#)
 - USART1.h, [140, 141](#)
 - USART2.h, [145, 146](#)
 - USART3.h, [150, 151](#)
- kSerial_5E2
 - USART0.h, [135, 136](#)
 - USART1.h, [140, 141](#)
 - USART2.h, [145, 146](#)
 - USART3.h, [150, 151](#)
- kSerial_5N1
 - USART0.h, [134–136](#)
 - USART1.h, [139–141](#)
 - USART2.h, [144–146](#)
 - USART3.h, [149–151](#)
- kSerial_5N2
 - USART0.h, [134–136](#)
 - USART1.h, [139–141](#)
 - USART2.h, [144–146](#)
 - USART3.h, [149–151](#)
- kSerial_5O1
 - USART0.h, [135, 136](#)
 - USART1.h, [140, 141](#)
 - USART2.h, [145, 146](#)
 - USART3.h, [150, 151](#)
- kSerial_5O2
 - USART0.h, [135–137](#)
 - USART1.h, [140–142](#)
 - USART2.h, [145–147](#)
 - USART3.h, [150–152](#)
- kSerial_6E1
 - USART0.h, [135, 136](#)
 - USART1.h, [140, 141](#)
 - USART2.h, [145, 146](#)
 - USART3.h, [150, 151](#)
- kSerial_6E2
 - USART0.h, [135, 136](#)
 - USART1.h, [140, 141](#)
 - USART2.h, [145, 146](#)
 - USART3.h, [150, 151](#)
- kSerial_6N1
 - USART0.h, [134–136](#)
 - USART1.h, [139–141](#)
 - USART2.h, [144–146](#)
 - USART3.h, [149–151](#)
- kSerial_6N2
 - USART0.h, [134–136](#)
 - USART1.h, [139–141](#)
 - USART2.h, [144–146](#)
 - USART3.h, [149–151](#)
- kSerial_6O1
 - USART0.h, [135, 136](#)
 - USART1.h, [140, 141](#)
 - USART2.h, [145, 146](#)
 - USART3.h, [150, 151](#)
- kSerial_6O2
 - USART0.h, [135–137](#)
 - USART1.h, [140–142](#)
 - USART2.h, [145–147](#)
 - USART3.h, [150–152](#)
- kSerial_7E1
 - USART0.h, [135, 136](#)
 - USART1.h, [140, 141](#)
 - USART2.h, [145, 146](#)
 - USART3.h, [150, 151](#)
- kSerial_7E2
 - USART0.h, [135, 136](#)
 - USART1.h, [140, 141](#)
 - USART2.h, [145, 146](#)

- USART3.h, 150, 151
- kSerial_7N1
 - USART0.h, 134–136
 - USART1.h, 139–141
 - USART2.h, 144–146
 - USART3.h, 149–151
- kSerial_7N2
 - USART0.h, 135, 136
 - USART1.h, 140, 141
 - USART2.h, 145, 146
 - USART3.h, 150, 151
- kSerial_7O1
 - USART0.h, 135–137
 - USART1.h, 140–142
 - USART2.h, 145–147
 - USART3.h, 150–152
- kSerial_7O2
 - USART0.h, 135–137
 - USART1.h, 140–142
 - USART2.h, 145–147
 - USART3.h, 150–152
- kSerial_8E1
 - USART0.h, 135, 136
 - USART1.h, 140, 141
 - USART2.h, 145, 146
 - USART3.h, 150, 151
- kSerial_8E2
 - USART0.h, 135, 136
 - USART1.h, 140, 141
 - USART2.h, 145, 146
 - USART3.h, 150, 151
- kSerial_8N1
 - USART0.h, 134–136
 - USART1.h, 139–141
 - USART2.h, 144–146
 - USART3.h, 149–151
- kSerial_8N2
 - USART0.h, 135, 136
 - USART1.h, 140, 141
 - USART2.h, 145, 146
 - USART3.h, 150, 151
- kSerial_8O1
 - USART0.h, 135–137
 - USART1.h, 140–142
 - USART2.h, 145–147
 - USART3.h, 150–152
- kSerial_8O2
 - USART0.h, 135–137
 - USART1.h, 140–142
 - USART2.h, 145–147
 - USART3.h, 150–152
- makeGpioVarFromGpioPin
 - GpioPinMacros.h, 107
- makeGpioVarFromGpioPinPwm
 - GpioPinMacros.h, 108
- MemUtils, 37
 - freeRam, 37
 - freeRamQuickEstimate, 37
- MemUtils.h, 117
- micros
 - SystemClock.h, 131
- millis
 - SystemClock.h, 131
- new.h, 117
- peek
 - Reader, 60
 - RingBuffer, 66
 - RingBufferT, 68
 - Serial0, 71
 - Serial1, 75
 - Serial2, 79
 - Serial3, 83
 - USART0, 38
 - USART1, 41
 - USART2, 44
 - USART3, 47
- print
 - Writer, 87–89
- println
 - Writer, 89–91
- processI2cMessage
 - I2cSlave, 35
- pull
 - RingBuffer, 66
 - RingBufferT, 68
- pullups
 - I2cMaster, 28
 - I2cSlave, 36
- push
 - RingBuffer, 66
 - RingBufferT, 69
- Pwm.h, 118
 - clearTimer0, 121
 - clearTimer1, 121
 - clearTimer2, 121
 - clearTimer3, 121
 - clearTimer4, 122
 - clearTimer5, 122
 - initPwmTimer0, 122
 - initPwmTimer1, 123
 - initPwmTimer2, 123
 - initPwmTimer3, 123
 - initPwmTimer4, 123
 - initPwmTimer5, 124

- writeGpioPinPwm, [120](#)
- writeGpioPinPwmV, [124](#)
- read
 - Reader, [61](#)
 - Serial0, [71](#)
 - Serial1, [75](#)
 - Serial2, [79](#)
 - Serial3, [83](#)
 - USART0, [39](#)
 - USART1, [42](#)
 - USART2, [45](#)
 - USART3, [48](#)
- readA2D
 - Analog2Digital.h, [98](#)
- readAsync
 - I2cMaster, [28](#), [29](#)
- readButtons
 - I2cLcd, [56](#)
- readBytes
 - Reader, [61](#)
- readBytesUntil
 - Reader, [61](#), [62](#)
- readFloat
 - Reader, [62](#)
- readGpioPinAnalog
 - Analog2Digital.h, [97](#)
- readGpioPinAnalogV
 - Analog2Digital.h, [98](#)
- readGpioPinDigital
 - GpioPinMacros.h, [108](#)
- readGpioPinDigitalV
 - GpioPinMacros.h, [109](#)
- readLine
 - Reader, [63](#)
- readLong
 - Reader, [63](#)
- readSync
 - I2cMaster, [29](#), [30](#)
- Reader, [58](#)
 - available, [59](#)
 - find, [59](#), [60](#)
 - findUntil, [60](#)
 - peek, [60](#)
 - read, [61](#)
 - readBytes, [61](#)
 - readBytesUntil, [61](#), [62](#)
 - readFloat, [62](#)
 - readLine, [63](#)
 - readLong, [63](#)
 - setTimeout, [64](#)
- Reader.h, [124](#)
- receiveUSART0
 - USARTMinimal.h, [154](#)
- receiveUSART1
 - USARTMinimal.h, [154](#)
- receiveUSART2
 - USARTMinimal.h, [155](#)
- receiveUSART3
 - USARTMinimal.h, [155](#)
- releaseUSART0
 - USARTMinimal.h, [155](#)
- releaseUSART1
 - USARTMinimal.h, [155](#)
- releaseUSART2
 - USARTMinimal.h, [156](#)
- releaseUSART3
 - USARTMinimal.h, [156](#)
- RingBuffer, [64](#)
 - isEmpty, [65](#)
 - isFull, [65](#)
 - isNotEmpty, [65](#)
 - isNotFull, [65](#)
 - peek, [66](#)
 - pull, [66](#)
 - push, [66](#)
 - RingBuffer, [65](#)
- RingBuffer.h, [125](#)
- RingBufferT
 - discardFromFront, [67](#)
 - isEmpty, [67](#)
 - isFull, [68](#)
 - isNotEmpty, [68](#)
 - isNotFull, [68](#)
 - peek, [68](#)
 - pull, [68](#)
 - push, [69](#)
- RingBufferT< T, N, SIZE >, [66](#)
- RingBufferT.h, [126](#)
- Serial0, [69](#)
 - available, [71](#)
 - peek, [71](#)
 - read, [71](#)
 - start, [71](#)
 - stop, [71](#)
 - write, [72](#)
- Serial1, [73](#)
 - available, [75](#)
 - peek, [75](#)
 - read, [75](#)
 - start, [75](#)
 - stop, [75](#)
 - write, [76](#)
- Serial2, [77](#)
 - available, [79](#)
 - peek, [79](#)
 - read, [79](#)

- start, [79](#)
- stop, [79](#)
- write, [80](#)
- Serial3, [81](#)
 - available, [83](#)
 - peek, [83](#)
 - read, [83](#)
 - start, [83](#)
 - stop, [83](#)
 - write, [84](#)
- setA2DVoltageReference
 - Analog2Digital.h, [98](#)
- setA2DVoltageReference11V
 - Analog2Digital.h, [99](#)
- setA2DVoltageReference256V
 - Analog2Digital.h, [99](#)
- setA2DVoltageReferenceAREF
 - Analog2Digital.h, [99](#)
- setA2DVoltageReferenceAVCC
 - Analog2Digital.h, [99](#)
- setBacklight
 - I2cLcd, [56](#)
- setCursor
 - I2cLcd, [56](#)
- setGpioPinHigh
 - GpioPinMacros.h, [108](#)
- setGpioPinHighV
 - GpioPinMacros.h, [109](#)
- setGpioPinLow
 - GpioPinMacros.h, [108](#)
- setGpioPinLowV
 - GpioPinMacros.h, [110](#)
- setGpioPinModeInput
 - GpioPinMacros.h, [108](#)
- setGpioPinModeInputPullup
 - GpioPinMacros.h, [109](#)
- setGpioPinModeInputPullupV
 - GpioPinMacros.h, [110](#)
- setGpioPinModeInputV
 - GpioPinMacros.h, [110](#)
- setGpioPinModeOutput
 - GpioPinMacros.h, [109](#)
- setGpioPinModeOutputV
 - GpioPinMacros.h, [110](#)
- setTimeout
 - Reader, [64](#)
- SimpleDelays.h, [127](#)
 - delayQuartersOfMicroSeconds, [128](#)
 - delayTenthsOfSeconds, [129](#)
 - delayWholeMilliSeconds, [129](#)
- start
 - I2cMaster, [30](#)
 - I2cSlave, [36](#)
 - Serial0, [71](#)
 - Serial1, [75](#)
 - Serial2, [79](#)
 - Serial3, [83](#)
 - USART0, [39](#)
 - USART1, [42](#)
 - USART2, [45](#)
 - USART3, [48](#)
- stop
 - I2cMaster, [30](#)
 - I2cSlave, [36](#)
 - Serial0, [71](#)
 - Serial1, [75](#)
 - Serial2, [79](#)
 - Serial3, [83](#)
 - USART0, [39](#)
 - USART1, [42](#)
 - USART2, [45](#)
 - USART3, [48](#)
- SystemClock.h, [129](#)
 - delay, [130](#)
 - delayMicroseconds, [130](#)
 - delayMilliseconds, [130](#)
 - initSystemClock, [130](#)
 - micros, [131](#)
 - millis, [131](#)
- transmitUSART0
 - USARTMinimal.h, [156](#)
- transmitUSART1
 - USARTMinimal.h, [156](#), [157](#)
- transmitUSART2
 - USARTMinimal.h, [157](#)
- transmitUSART3
 - USARTMinimal.h, [157](#), [158](#)
- USART0, [37](#)
 - available, [38](#)
 - flush, [38](#)
 - peek, [38](#)
 - read, [39](#)
 - start, [39](#)
 - stop, [39](#)
 - write, [39](#), [40](#)
- USART0.h, [131](#)
 - kSerial_5E1, [135](#), [136](#)
 - kSerial_5E2, [135](#), [136](#)
 - kSerial_5N1, [134–136](#)
 - kSerial_5N2, [134–136](#)
 - kSerial_5O1, [135](#), [136](#)
 - kSerial_5O2, [135–137](#)
 - kSerial_6E1, [135](#), [136](#)
 - kSerial_6E2, [135](#), [136](#)
 - kSerial_6N1, [134–136](#)
 - kSerial_6N2, [134–136](#)
 - kSerial_6O1, [135](#), [136](#)

- kSerial_6O2, 135–137
- kSerial_7E1, 135, 136
- kSerial_7E2, 135, 136
- kSerial_7N1, 134–136
- kSerial_7N2, 135, 136
- kSerial_7O1, 135–137
- kSerial_7O2, 135–137
- kSerial_8E1, 135, 136
- kSerial_8E2, 135, 136
- kSerial_8N1, 134–136
- kSerial_8N2, 135, 136
- kSerial_8O1, 135–137
- kSerial_8O2, 135–137
- UsartSerialConfiguration, 134
- USART1, 40
 - available, 41
 - flush, 41
 - peek, 41
 - read, 42
 - start, 42
 - stop, 42
 - write, 42, 43
- USART1.h, 137
 - kSerial_5E1, 140, 141
 - kSerial_5E2, 140, 141
 - kSerial_5N1, 139–141
 - kSerial_5N2, 139–141
 - kSerial_5O1, 140, 141
 - kSerial_5O2, 140–142
 - kSerial_6E1, 140, 141
 - kSerial_6E2, 140, 141
 - kSerial_6N1, 139–141
 - kSerial_6N2, 139–141
 - kSerial_6O1, 140, 141
 - kSerial_6O2, 140–142
 - kSerial_7E1, 140, 141
 - kSerial_7E2, 140, 141
 - kSerial_7N1, 139–141
 - kSerial_7N2, 140, 141
 - kSerial_7O1, 140–142
 - kSerial_7O2, 140–142
 - kSerial_8E1, 140, 141
 - kSerial_8E2, 140, 141
 - kSerial_8N1, 139–141
 - kSerial_8N2, 140, 141
 - kSerial_8O1, 140–142
 - kSerial_8O2, 140–142
 - UsartSerialConfiguration, 139
- USART2, 43
 - available, 44
 - flush, 44
 - peek, 44
 - read, 45
 - start, 45
 - stop, 45
 - write, 45, 46
- USART2.h, 142
 - kSerial_5E1, 145, 146
 - kSerial_5E2, 145, 146
 - kSerial_5N1, 144–146
 - kSerial_5N2, 144–146
 - kSerial_5O1, 145, 146
 - kSerial_5O2, 145–147
 - kSerial_6E1, 145, 146
 - kSerial_6E2, 145, 146
 - kSerial_6N1, 144–146
 - kSerial_6N2, 144–146
 - kSerial_6O1, 145, 146
 - kSerial_6O2, 145–147
 - kSerial_7E1, 145, 146
 - kSerial_7E2, 145, 146
 - kSerial_7N1, 144–146
 - kSerial_7N2, 145, 146
 - kSerial_7O1, 145–147
 - kSerial_7O2, 145–147
 - kSerial_8E1, 145, 146
 - kSerial_8E2, 145, 146
 - kSerial_8N1, 144–146
 - kSerial_8N2, 145, 146
 - kSerial_8O1, 145–147
 - kSerial_8O2, 145–147
 - UsartSerialConfiguration, 144
- USART3, 46
 - available, 47
 - flush, 47
 - peek, 47
 - read, 48
 - start, 48
 - stop, 48
 - write, 48, 49
- USART3.h, 147
 - kSerial_5E1, 150, 151
 - kSerial_5E2, 150, 151
 - kSerial_5N1, 149–151
 - kSerial_5N2, 149–151
 - kSerial_5O1, 150, 151
 - kSerial_5O2, 150–152
 - kSerial_6E1, 150, 151
 - kSerial_6E2, 150, 151
 - kSerial_6N1, 149–151
 - kSerial_6N2, 149–151
 - kSerial_6O1, 150, 151
 - kSerial_6O2, 150–152
 - kSerial_7E1, 150, 151
 - kSerial_7E2, 150, 151
 - kSerial_7N1, 149–151
 - kSerial_7N2, 150, 151
 - kSerial_7O1, 150–152

- kSerial_7O2, [150–152](#)
- kSerial_8E1, [150](#), [151](#)
- kSerial_8E2, [150](#), [151](#)
- kSerial_8N1, [149–151](#)
- kSerial_8N2, [150](#), [151](#)
- kSerial_8O1, [150–152](#)
- kSerial_8O2, [150–152](#)
- UsartSerialConfiguration, [149](#)
- USARTMinimal.h, [152](#)
 - initUSART0, [153](#)
 - initUSART1, [153](#)
 - initUSART2, [154](#)
 - initUSART3, [154](#)
 - receiveUSART0, [154](#)
 - receiveUSART1, [154](#)
 - receiveUSART2, [155](#)
 - receiveUSART3, [155](#)
 - releaseUSART0, [155](#)
 - releaseUSART1, [155](#)
 - releaseUSART2, [156](#)
 - releaseUSART3, [156](#)
 - transmitUSART0, [156](#)
 - transmitUSART1, [156](#), [157](#)
 - transmitUSART2, [157](#)
 - transmitUSART3, [157](#), [158](#)
- UsartSerialConfiguration
 - USART0.h, [134](#)
 - USART1.h, [139](#)
 - USART2.h, [144](#)
 - USART3.h, [149](#)
- write
 - I2cLcd, [56](#), [57](#)
 - Serial0, [72](#)
 - Serial1, [76](#)
 - Serial2, [80](#)
 - Serial3, [84](#)
 - USART0, [39](#), [40](#)
 - USART1, [42](#), [43](#)
 - USART2, [45](#), [46](#)
 - USART3, [48](#), [49](#)
 - Writer, [92](#)
- writeAsync
 - I2cMaster, [30–32](#)
- writeGpioPinDigital
 - GpioPinMacros.h, [109](#)
- writeGpioPinDigitalV
 - GpioPinMacros.h, [110](#)
- writeGpioPinPwm
 - Pwm.h, [120](#)
- writeGpioPinPwmV
 - Pwm.h, [124](#)
- writeSync
 - I2cMaster, [32](#), [33](#)
- Writer, [85](#)
 - IntegerOutputBase, [87](#)
 - kBin, [87](#)
 - kDec, [87](#)
 - kHex, [87](#)
 - kOct, [87](#)
 - print, [87–89](#)
 - println, [89–91](#)
 - write, [92](#)
- Writer.h, [158](#)