

Murasaki Class Library  
3.0.0

Generated by Doxygen 1.8.11

## Contents

<b>1</b>	<b>Preface</b>	<b>1</b>
1.1	Simplified IO . . . . .	2
1.2	Preemptive multi-task . . . . .	2
1.3	Synchronous IO . . . . .	2
1.4	Thread-safe IO . . . . .	2
1.5	Versatile printf() logger . . . . .	3
1.6	Guard by assertion . . . . .	3
1.7	System Logging . . . . .	3
1.8	Configurable . . . . .	3
<b>2</b>	<b>Target and Environment</b>	<b>4</b>
<b>3</b>	<b>Usage Introduction</b>	<b>4</b>
3.1	Message output . . . . .	5
3.2	Serial communication . . . . .	5
3.3	Debugging with Murasaki. . . . .	6
3.4	Tasking . . . . .	7
3.5	Other peripherals . . . . .	8
3.5.1	I2C Master . . . . .	8
3.5.2	I2C Slave . . . . .	8
3.5.3	SPI Master . . . . .	9
3.5.4	SPI Slave . . . . .	9
3.5.5	GPIO . . . . .	10
3.5.6	Duplex Audio . . . . .	10
3.6	Program flow . . . . .	11
3.6.1	Application flow . . . . .	11
3.6.2	HAL Assertion flow . . . . .	13
3.6.3	Spurious Interrupt flow . . . . .	13
3.6.4	Assertion flow . . . . .	14
3.6.5	General Interrupt flow . . . . .	14
3.6.6	EXTI flow . . . . .	14

<b>4</b>	<b>Porting guide</b>	<b>15</b>
4.1	Directory Structure	15
4.1.1	Src directory	15
4.1.2	Inc directory	15
4.1.3	Src/Thirdparty and Inc/Thirdparty directory	15
4.1.4	murasaki.hpp	16
4.1.5	template directory	16
4.1.6	install script	17
4.2	CubelIDE setting	17
4.2.1	Heap Size	18
4.2.2	Stack Size	19
4.2.3	Task stack size of the default task	20
4.2.4	UART peripheral	20
4.2.5	SPI Master peripheral	20
4.2.6	SPI Slave peripheral	20
4.2.7	I2C peripheral	21
4.2.8	EXTI	21
4.3	Configuration	21
4.4	Task Priority and Stack Size	21
4.5	Heap memory consideration	21
4.6	Platform variable	22
4.7	Routing interrupts	23
4.8	Error handling	24
4.9	Summary of the porting	25

<b>5</b>	<b>Module Documentation</b>	<b>26</b>
5.1	Murasaki API reference	26
5.1.1	Detailed Description	27
5.2	Murasaki Class Collection	28
5.2.1	Detailed Description	29
5.2.2	Macro Definition Documentation	29
5.3	Third party classes	32
5.3.1	Detailed Description	32
5.4	Definitions and Configuration	33
5.4.1	Detailed Description	33
5.4.2	Macro Definition Documentation	33
5.4.3	Enumeration Type Documentation	35
5.5	Application Specific Platform	40
5.5.1	Detailed Description	41
5.5.2	Function Documentation	41
5.5.3	Variable Documentation	46
5.6	Abstract Classes	47
5.6.1	Detailed Description	48
5.7	Synchronization and Exclusive access	49
5.7.1	Detailed Description	49
5.8	Helper classes	50
5.8.1	Detailed Description	50
5.8.2	Function Documentation	50
5.9	Utility functions	52
5.9.1	Detailed Description	52
5.9.2	Function Documentation	52
5.10	CMSIS	54
5.10.1	Detailed Description	54
5.11	Stm32f4xx_system	55
5.11.1	Detailed Description	55

5.12	STM32F4xx_System_Private_Includes	56
5.12.1	Detailed Description	56
5.12.2	Macro Definition Documentation	56
5.13	STM32F4xx_System_Private_TypesDefinitions	57
5.14	STM32F4xx_System_Private_Defines	58
5.14.1	Detailed Description	58
5.14.2	Macro Definition Documentation	58
5.15	STM32F4xx_System_Private_Macros	59
5.16	STM32F4xx_System_Private_Variables	60
5.16.1	Detailed Description	60
5.17	STM32F4xx_System_Private_FunctionPrototypes	61
5.18	STM32F4xx_System_Private_Functions	62
5.18.1	Detailed Description	62
5.18.2	Function Documentation	62
<b>6</b>	<b>Namespace Documentation</b>	<b>64</b>
6.1	murasaki Namespace Reference	64
6.1.1	Detailed Description	66
6.1.2	Function Documentation	66
6.1.3	Variable Documentation	68
<b>7</b>	<b>Class Documentation</b>	<b>68</b>
7.1	murasaki::Adau1361 Class Reference	68
7.1.1	Detailed Description	70
7.1.2	Constructor & Destructor Documentation	70
7.1.3	Member Function Documentation	71
7.2	murasaki::Adc Class Reference	74
7.2.1	Detailed Description	75
7.2.2	Constructor & Destructor Documentation	76
7.2.3	Member Function Documentation	76
7.3	murasaki::AdcStrategy Class Reference	78

7.3.1	Member Function Documentation	79
7.4	<a href="#">murasaki::AudioCodecStrategy Class Reference</a>	81
7.4.1	Detailed Description	82
7.4.2	Constructor & Destructor Documentation	82
7.4.3	Member Function Documentation	82
7.5	<a href="#">murasaki::AudioPortAdapterStrategy Class Reference</a>	83
7.5.1	Detailed Description	84
7.5.2	Member Function Documentation	85
7.6	<a href="#">murasaki::BitIn Class Reference</a>	88
7.6.1	Detailed Description	89
7.6.2	Constructor & Destructor Documentation	90
7.6.3	Member Function Documentation	90
7.7	<a href="#">murasaki::BitInStrategy Class Reference</a>	91
7.7.1	Detailed Description	91
7.7.2	Member Function Documentation	92
7.8	<a href="#">murasaki::BitOut Class Reference</a>	92
7.8.1	Detailed Description	93
7.8.2	Constructor & Destructor Documentation	94
7.8.3	Member Function Documentation	95
7.9	<a href="#">murasaki::BitOutStrategy Class Reference</a>	95
7.9.1	Detailed Description	96
7.9.2	Member Function Documentation	97
7.10	<a href="#">murasaki::CallbackRepositorySingleton Class Reference</a>	97
7.10.1	Detailed Description	98
7.10.2	Member Function Documentation	98
7.11	<a href="#">murasaki::CriticalSection Class Reference</a>	99
7.11.1	Detailed Description	99
7.11.2	Member Function Documentation	99
7.12	<a href="#">murasaki::Debugger Class Reference</a>	100
7.12.1	Detailed Description	101

7.12.2	Constructor & Destructor Documentation . . . . .	101
7.12.3	Member Function Documentation . . . . .	101
7.12.4	Member Data Documentation . . . . .	102
7.13	murasaki::DebuggerFifo Class Reference . . . . .	103
7.13.1	Detailed Description . . . . .	104
7.13.2	Constructor & Destructor Documentation . . . . .	104
7.13.3	Member Function Documentation . . . . .	104
7.14	murasaki::DebuggerUart Class Reference . . . . .	105
7.14.1	Detailed Description . . . . .	106
7.14.2	Constructor & Destructor Documentation . . . . .	106
7.14.3	Member Function Documentation . . . . .	107
7.15	murasaki::DuplexAudio Class Reference . . . . .	110
7.15.1	Detailed Description . . . . .	111
7.15.2	Constructor & Destructor Documentation . . . . .	111
7.15.3	Member Function Documentation . . . . .	112
7.16	murasaki::Exti Class Reference . . . . .	115
7.16.1	Detailed Description . . . . .	116
7.16.2	Constructor & Destructor Documentation . . . . .	118
7.16.3	Member Function Documentation . . . . .	119
7.17	murasaki::ExtiCallbackRepositorySingleton Class Reference . . . . .	120
7.17.1	Detailed Description . . . . .	121
7.17.2	Member Function Documentation . . . . .	121
7.18	murasaki::FifoStrategy Class Reference . . . . .	122
7.18.1	Detailed Description . . . . .	122
7.18.2	Constructor & Destructor Documentation . . . . .	122
7.18.3	Member Function Documentation . . . . .	123
7.19	murasaki::GPIO_type Struct Reference . . . . .	123
7.19.1	Detailed Description . . . . .	124
7.20	murasaki::I2cCallbackStrategy Class Reference . . . . .	124
7.20.1	Detailed Description . . . . .	125

7.20.2	Member Function Documentation	125
7.21	murasaki::I2cMaster Class Reference	126
7.21.1	Detailed Description	127
7.21.2	Constructor & Destructor Documentation	129
7.21.3	Member Function Documentation	129
7.22	murasaki::I2cMasterStrategy Class Reference	133
7.22.1	Detailed Description	134
7.22.2	Member Function Documentation	134
7.23	murasaki::I2cSlave Class Reference	136
7.23.1	Detailed Description	137
7.23.2	Member Function Documentation	137
7.24	murasaki::I2cSlaveStrategy Class Reference	140
7.24.1	Detailed Description	141
7.24.2	Member Function Documentation	141
7.25	murasaki::I2sPortAdapter Class Reference	142
7.25.1	Detailed Description	144
7.25.2	Constructor & Destructor Documentation	146
7.25.3	Member Function Documentation	146
7.26	murasaki::InterruptStrategy Class Reference	150
7.26.1	Detailed Description	150
7.26.2	Member Function Documentation	150
7.27	murasaki::LoggerStrategy Class Reference	151
7.27.1	Detailed Description	152
7.27.2	Constructor & Destructor Documentation	152
7.27.3	Member Function Documentation	152
7.28	murasaki::LoggingHelpers Struct Reference	153
7.29	murasaki::PeripheralStrategy Class Reference	154
7.29.1	Detailed Description	155
7.29.2	Member Function Documentation	155
7.30	murasaki::Platform Struct Reference	156



7.30.1 Detailed Description . . . . .	157
7.31 <a href="#">murasaki::QuadratureEncoder Class Reference</a> . . . . .	157
7.31.1 Detailed Description . . . . .	158
7.31.2 Constructor & Destructor Documentation . . . . .	158
7.31.3 Member Function Documentation . . . . .	159
7.32 <a href="#">murasaki::QuadratureEncoderStrategy Class Reference</a> . . . . .	159
7.32.1 Detailed Description . . . . .	160
7.32.2 Member Function Documentation . . . . .	161
7.33 <a href="#">murasaki::SaiPortAdapter Class Reference</a> . . . . .	161
7.33.1 Detailed Description . . . . .	163
7.33.2 Constructor & Destructor Documentation . . . . .	165
7.33.3 Member Function Documentation . . . . .	166
7.34 <a href="#">murasaki::SimpleTask Class Reference</a> . . . . .	169
7.34.1 Detailed Description . . . . .	170
7.34.2 Constructor & Destructor Documentation . . . . .	170
7.34.3 Member Function Documentation . . . . .	171
7.35 <a href="#">murasaki::SpiCallbackStrategy Class Reference</a> . . . . .	171
7.35.1 Member Function Documentation . . . . .	172
7.36 <a href="#">murasaki::SpiMaster Class Reference</a> . . . . .	173
7.36.1 Detailed Description . . . . .	174
7.36.2 Constructor & Destructor Documentation . . . . .	176
7.36.3 Member Function Documentation . . . . .	176
7.37 <a href="#">murasaki::SpiMasterStrategy Class Reference</a> . . . . .	178
7.37.1 Detailed Description . . . . .	179
7.37.2 Member Function Documentation . . . . .	179
7.38 <a href="#">murasaki::SpiSlave Class Reference</a> . . . . .	180
7.38.1 Detailed Description . . . . .	181
7.38.2 Constructor & Destructor Documentation . . . . .	183
7.38.3 Member Function Documentation . . . . .	183
7.39 <a href="#">murasaki::SpiSlaveAdapter Class Reference</a> . . . . .	185

7.39.1 Detailed Description . . . . .	186
7.39.2 Constructor & Destructor Documentation . . . . .	186
7.39.3 Member Function Documentation . . . . .	187
7.40 murasaki::SpiSlaveAdapterStrategy Class Reference . . . . .	187
7.40.1 Detailed Description . . . . .	188
7.40.2 Constructor & Destructor Documentation . . . . .	188
7.40.3 Member Function Documentation . . . . .	189
7.41 murasaki::SpiSlaveStrategy Class Reference . . . . .	190
7.41.1 Detailed Description . . . . .	191
7.41.2 Member Function Documentation . . . . .	191
7.42 murasaki::Synchronizer Class Reference . . . . .	191
7.42.1 Detailed Description . . . . .	192
7.42.2 Member Function Documentation . . . . .	192
7.43 murasaki::TaskStrategy Class Reference . . . . .	192
7.43.1 Detailed Description . . . . .	193
7.43.2 Constructor & Destructor Documentation . . . . .	193
7.43.3 Member Function Documentation . . . . .	194
7.44 murasaki::Uart Class Reference . . . . .	195
7.44.1 Detailed Description . . . . .	196
7.44.2 Constructor & Destructor Documentation . . . . .	198
7.44.3 Member Function Documentation . . . . .	198
7.45 murasaki::UartLogger Class Reference . . . . .	203
7.45.1 Detailed Description . . . . .	204
7.45.2 Constructor & Destructor Documentation . . . . .	204
7.45.3 Member Function Documentation . . . . .	204
7.46 murasaki::UartStrategy Class Reference . . . . .	205
7.46.1 Detailed Description . . . . .	206
7.46.2 Member Function Documentation . . . . .	206

<b>8</b>	<b>File Documentation</b>	<b>209</b>
8.1	adau1361.hpp File Reference . . . . .	209
8.1.1	Detailed Description . . . . .	210
8.2	adc.hpp File Reference . . . . .	210
8.2.1	Detailed Description . . . . .	211
8.3	adcstrategy.hpp File Reference . . . . .	212
8.3.1	Detailed Description . . . . .	213
8.4	allocators.cpp File Reference . . . . .	213
8.4.1	Detailed Description . . . . .	214
8.5	audiocodecstrategy.hpp File Reference . . . . .	214
8.5.1	Detailed Description . . . . .	215
8.6	audioportadapterstrategy.hpp File Reference . . . . .	216
8.6.1	Detailed Description . . . . .	217
8.7	bitin.hpp File Reference . . . . .	217
8.7.1	Detailed Description . . . . .	218
8.8	bitinstrategy.hpp File Reference . . . . .	218
8.8.1	Detailed Description . . . . .	220
8.9	bitout.hpp File Reference . . . . .	220
8.9.1	Detailed Description . . . . .	221
8.10	bitoutstrategy.hpp File Reference . . . . .	222
8.10.1	Detailed Description . . . . .	223
8.11	callbackrepositorysingleton.hpp File Reference . . . . .	224
8.11.1	Detailed Description . . . . .	225
8.12	criticalsection.hpp File Reference . . . . .	225
8.12.1	Detailed Description . . . . .	226
8.13	debugger.hpp File Reference . . . . .	226
8.13.1	Detailed Description . . . . .	227
8.14	debuggerfifo.hpp File Reference . . . . .	228
8.14.1	Detailed Description . . . . .	229
8.15	debuggeruart.hpp File Reference . . . . .	230

8.15.1 Detailed Description . . . . .	231
8.16 duplexaudio.hpp File Reference . . . . .	231
8.16.1 Detailed Description . . . . .	232
8.17 exti.hpp File Reference . . . . .	232
8.17.1 Detailed Description . . . . .	234
8.18 exticallbackrepositorysingleton.hpp File Reference . . . . .	234
8.18.1 Detailed Description . . . . .	235
8.19 fifostrategy.hpp File Reference . . . . .	235
8.19.1 Detailed Description . . . . .	236
8.20 i2ccallbackstrategy.hpp File Reference . . . . .	237
8.20.1 Detailed Description . . . . .	238
8.21 i2cmaster.hpp File Reference . . . . .	239
8.21.1 Detailed Description . . . . .	240
8.22 i2cmasterstrategy.hpp File Reference . . . . .	241
8.22.1 Detailed Description . . . . .	242
8.23 i2cslave.hpp File Reference . . . . .	243
8.23.1 Detailed Description . . . . .	244
8.24 i2cslavestrategy.hpp File Reference . . . . .	244
8.24.1 Detailed Description . . . . .	245
8.25 i2sportadapter.hpp File Reference . . . . .	246
8.25.1 Detailed Description . . . . .	247
8.26 interruptstrategy.hpp File Reference . . . . .	247
8.26.1 Detailed Description . . . . .	248
8.27 loggerstrategy.hpp File Reference . . . . .	249
8.27.1 Detailed Description . . . . .	250
8.28 main.c File Reference . . . . .	251
8.28.1 Detailed Description . . . . .	251
8.28.2 Function Documentation . . . . .	252
8.28.3 Variable Documentation . . . . .	253
8.29 main.h File Reference . . . . .	254

8.29.1 Detailed Description . . . . .	254
8.29.2 Function Documentation . . . . .	255
8.30 murasaki.hpp File Reference . . . . .	255
8.30.1 Detailed Description . . . . .	256
8.31 murasaki_0_intro.hpp File Reference . . . . .	256
8.31.1 Detailed Description . . . . .	256
8.32 murasaki_1_env.hpp File Reference . . . . .	256
8.32.1 Detailed Description . . . . .	257
8.33 murasaki_2_ug.hpp File Reference . . . . .	257
8.33.1 Detailed Description . . . . .	257
8.34 murasaki_3_pg.hpp File Reference . . . . .	257
8.34.1 Detailed Description . . . . .	257
8.35 murasaki_4_mod.hpp File Reference . . . . .	257
8.35.1 Detailed Description . . . . .	257
8.36 murasaki_assert.hpp File Reference . . . . .	258
8.36.1 Detailed Description . . . . .	259
8.37 murasaki_callback.cpp File Reference . . . . .	259
8.37.1 Detailed Description . . . . .	260
8.37.2 Function Documentation . . . . .	260
8.38 murasaki_config.hpp File Reference . . . . .	262
8.38.1 Detailed Description . . . . .	263
8.39 murasaki_defs.hpp File Reference . . . . .	263
8.39.1 Detailed Description . . . . .	264
8.40 murasaki_include_stub.h File Reference . . . . .	264
8.40.1 Detailed Description . . . . .	265
8.41 murasaki_platform.cpp File Reference . . . . .	265
8.41.1 Detailed Description . . . . .	266
8.42 murasaki_platform.hpp File Reference . . . . .	266
8.42.1 Detailed Description . . . . .	267
8.42.2 Function Documentation . . . . .	267

8.43	<a href="#">murasaki_syslog.hpp File Reference</a>	268
8.43.1	<a href="#">Detailed Description</a>	269
8.44	<a href="#">peripheralstrategy.hpp File Reference</a>	269
8.44.1	<a href="#">Detailed Description</a>	270
8.45	<a href="#">platform_config.hpp File Reference</a>	270
8.45.1	<a href="#">Detailed Description</a>	271
8.45.2	<a href="#">Macro Definition Documentation</a>	271
8.46	<a href="#">platform_defs.hpp File Reference</a>	271
8.46.1	<a href="#">Detailed Description</a>	272
8.47	<a href="#">quadratureencoder.cpp File Reference</a>	272
8.47.1	<a href="#">Detailed Description</a>	273
8.48	<a href="#">quadratureencoder.hpp File Reference</a>	273
8.48.1	<a href="#">Detailed Description</a>	274
8.49	<a href="#">quadratureencoderstrategy.hpp File Reference</a>	274
8.49.1	<a href="#">Detailed Description</a>	276
8.50	<a href="#">saiportadapter.hpp File Reference</a>	276
8.50.1	<a href="#">Detailed Description</a>	277
8.51	<a href="#">simpletask.hpp File Reference</a>	277
8.51.1	<a href="#">Detailed Description</a>	279
8.52	<a href="#">spicallbackstrategy.hpp File Reference</a>	279
8.52.1	<a href="#">Detailed Description</a>	280
8.53	<a href="#">spimaster.hpp File Reference</a>	281
8.53.1	<a href="#">Detailed Description</a>	282
8.54	<a href="#">spimasterstrategy.hpp File Reference</a>	282
8.54.1	<a href="#">Detailed Description</a>	283
8.55	<a href="#">spislave.hpp File Reference</a>	284
8.55.1	<a href="#">Detailed Description</a>	285
8.56	<a href="#">spislaveadapter.hpp File Reference</a>	285
8.56.1	<a href="#">Detailed Description</a>	286
8.57	<a href="#">spislaveadapterstrategy.hpp File Reference</a>	286

8.57.1 Detailed Description . . . . .	288
8.58 spislavestrategy.hpp File Reference . . . . .	288
8.58.1 Detailed Description . . . . .	289
8.59 stm32f4xx_it.c File Reference . . . . .	290
8.59.1 Detailed Description . . . . .	291
8.59.2 Variable Documentation . . . . .	291
8.60 stm32f4xx_it.h File Reference . . . . .	292
8.60.1 Detailed Description . . . . .	293
8.61 synchronizer.hpp File Reference . . . . .	294
8.61.1 Detailed Description . . . . .	295
8.62 system_stm32f4xx.c File Reference . . . . .	295
8.62.1 Detailed Description . . . . .	296
8.63 taskstrategy.hpp File Reference . . . . .	297
8.63.1 Detailed Description . . . . .	298
8.64 uart.hpp File Reference . . . . .	298
8.64.1 Detailed Description . . . . .	299
8.65 uartlogger.hpp File Reference . . . . .	299
8.65.1 Detailed Description . . . . .	301
8.66 uartstrategy.hpp File Reference . . . . .	301
8.66.1 Detailed Description . . . . .	302
<b>Index</b>	<b>303</b>

## 1 Preface

Murasaki is a class library on the STM32Cube HAL and FreeRTOS. By using Murasaki, you can program STM32 series quickly and easily. You can obtain the source code of the Murasaki Library from the [GitHub repository](#).

Murasaki has following design philosophies:

- Simplified IO
- Preemptive multi-task
- Synchronous IO
- Thread-safe IO
- Versatile printf() logger
- Guard by assertion
- System Logging
- Configurable

There are some other manuals of murasaki class library :

- Usage Introduction
- Porting guide
- Murasaki Class Collection

## 1.1 Simplified IO

The class types package the IO functions. For example, The `murasaki::Uart` class can receive a UART handle.

```
murasaki::UartStrategy * uart3 = new murasaki::Uart( &huart3 );
```

Where `huart3` is a UART port 3 handle, which is generated by the CubeIDE.

The STM32Cube HAL is quite rich and flexible. On the other hand, it is quite large and complex. The classes in Murasaki simplify it by letting flexibility beside. For example, the `murasaki::Uart` class can support only the DMA transfer. The polling-based transfer is not supported. By giving up the flexibility, programming with Murasaki is more comfortable than using HAL directly.

## 1.2 Preemptive multi-task

The Murasaki class library is built on FreeRTOS's preemptive configuration. As a result, Murasaki is automatically aware of preemptive multi-task. That means, Murasaki's classes don't use polling to wait for any event. Thus, a task can do some job while other tasks are waiting for the end of the IO operation.

The multi-task programming helps to divide a big program into sub-units. This dividing is an excellent way to develop large programs easier. And the more critical point, it is easier to maintain.

## 1.3 Synchronous IO

The synchronous IO is one of the most critical features of Murasaki. The peripheral wrapping class like `murasaki::Uart` provides a set of member functions to do the data transmission/receiving. Such the member functions are programmed as "synchronous" IO.

The synchronous IO function doesn't return until each IO function finished. For example, if you transmit 10bytes through the UART, the IO member function transmits the 10bytes data, and then, return.

Note: Sometimes, the "completion" means the end of the DMA transfer session, rather than the accurate transmission of the last byte. In this case, the system generates a completion interrupt while the data is still in FIFO of the peripheral. Anyway, this is a hardware issue.

Some member functions are restricted to use only in the task context to allow the synchronous and blocking IO.

## 1.4 Thread-safe IO

The synchronous IO and the preemptive multi-task provide more accessible programming. On the other hand, there is a possibility that two different tasks access one peripheral simultaneously. This kind of access messes the peripheral's behavior.

To prevent this condition, some peripheral wrapping class has an exclusive access mechanism by a mutex.

By this mechanism, if two tasks try to transmit through one peripheral, one task is kept waiting until the other finished to transmit. This behavior is called blocking.



## 1.5 Versatile printf() logger

Logging or "printf debug" is a strong tool in the embedded system development. Murasaki has three levels of the printf debugging mechanism. One is the `murasaki::debugger->Printf()`, the second is `MURASAKI_ASSERT` macro. In addition to these two, `MURASAKI_SYSLOG` macro is available.

The `murasaki::debugger->Printf()` is flexible output mechanism which has several good features :

- printf() compatible parameters.
- Task/interrupt bi-context operation
- None-blocking logging by internal FIFO.
- User configurable output port

These features allow a programmer to do the printf() debug not only in the task context but also in the interrupt context.

## 1.6 Guard by assertion

In addition to the `murasaki::debugger->Printf()`, programmer can use `MURASAKI_ASSERT` macro. This macro allows for easy assertion and logging. This macro uses the `murasaki::debugger->Printf()` internally.

If the assertion failed, a message would be output to the debug port with the information of the source line number and file name.

Murasaki class library is using the assertion widely. As a result, the wrong context, wrong parameter, etc., will be reported to the debugger output.

## 1.7 System Logging

`MURASAKI_SYSLOG` provides the message output based on the level and filtering. This mechanism is intended to help the Murasaki library development. But also the application can use this mechanism.

## 1.8 Configurable

Murasaki can configure its basic parameter by customer. For example, you can change the size of the printf buffer.

## 2 Target and Environment

Murasaki library is developed with following environment:

- [STM32CubeIDE 1.3.0](#)
- [Ubuntu 16.04 \(64bit\)](#)

And then, confirmed portability with following boards :

- [Nucleo H743ZI \( STM32H743ZI : Cortex-M7 \)](#)
- [Nucleo F746ZG \( STM32F746ZG : Cortex-M7 \)](#)
- [Nucleo F722ZE \( STM32F722ZE : Cortex-M7 \)](#)
- [Nucleo F446RE \( STM32F446RE : Cortex-M4 \)](#)
- [Nucleo G431RB \( STM32G431RB : Cortex-M4 \)](#)
- [Nucleo L412RB-P \( STM32L412RB : Cortex-M4 \)](#)
- [Nucleo L152RE \( STM32L152RE : Cortex-M3 \)](#)
- [Nucleo F091RC \( STM32F091RC : Cortex-M0 \)](#)
- [Nucleo G070RB \( STM32G070RB : Cortex-M0+ \)](#)

## 3 Usage Introduction

In this introduction, we see how to use the Murasaki class library in the STM32 program.

- [Message output](#)
- [Serial communication](#)
- [Debugging with Murasaki.](#)
- [Tasking](#)
- [Other peripherals](#)
- [Program flow](#)

There are some other manuals of murasaki class library :

- [Preface](#)
- [Porting guide](#)
- [Murasaki Class Collection](#)

For the easy-to-understand description, we assume several things on the application skeleton, which we are going to use Murasaki :

- The application skeleton is generated by [CubeIDE](#)
- The application skeleton is configured to use FreeRTOS
- UART3 is configured to use DMA.
- UART3 is configured to use interrupt.

These are requirements from the Murasaki library.

### 3.1 Message output

The Murasaki library has a `Printf()` like a message output mechanism. This mechanism is an easy way to display a message from an embedded microcomputer to the terminal simulator like Kermit on a host computer. Murasaki's `Printf()` is based on the standard C language formatting library. So, a programmer can output a message like standard `printf()`.

As usual, let's start from "hello, world."

```
murasaki::debugger->Printf("Hello, world!\n");
```

In Murasaki manner, the `Printf()` is not a global function. This member function is a method of `murasaki::Debugger` class. The `murasaki::debugger` variable is one of the two Murasaki's global variables. And it provides an easy to use message output.

The end-of-line character depends on the terminal. In the above example, the terminator is '  
'.

The '  
' is for the Linux based Kermit software. Other terminal systems may need other end-of-line characters. The easiest way to absorb this difference is to change the terminal software behavior. Almost terminal emulation software can accept any type of end-of-line character.

Because the `Printf()` works as like standard `printf()`, you can also use the format string.

```
murasaki::debugger->Printf("count is %d\n", count);
```

The `Printf()` is designed as a debugger message output for an embedded real-time system. This function is :

- Thread safe
- Asynchronous
- Blocking
- Buffered

In other words, you can use this function in either task or interrupt handler without bothering the real-time process.

### 3.2 Serial communication

`murasaki::Uart` is the asynchronous serial communication. The initial baud rate, parity and data size are defined by CubeIDE. So, there is no need to initialize the communication parameter in application program. User can transmit data by passing its buffer address and data length.

```
uint8_t data[5] = { 1, 2, 3, 4, 5 };  
murasaki::UartStatus stat;  
  
stat = murasaki::platform.uart->Transmit(  
    data,  
    5);
```

In addition to the transmit function, also `Receive()` member function exists.

### 3.3 Debugging with Murasaki.

As we saw again and again, Murasaki has a simple messaging output for real-time debugging. This feature is typically used as UART serial output, but configurable by the programmer.

The `murasaki::debugger` is the useful variable to output the debugging message. There are several good features in this function, as we already saw.

- Versatile `printf()` style format string.
- Can call from both task and interrupt context
- Asynchronous
- Non-blocking

These features help the programmer to display the message in the real-time, multi-task application.

In addition to this simple debugging variable, a programmer can use `assert_failure()` function of the STM32 HAL. The STM32Cube HAL has `assert_failure()` to check the parameter on the fly. By default, this function is disabled. To use this function, programmer have to make it enable, and add function to receive the debug information.

To enable the `assert_failuer()`, open the Project Management tab of the Device Configuration Tool of the CubeIDE. And then, go to the Code Generation section and check "Enable Full Assert".

And then, you must modify `assert_failure()` in `main.c`, to call output function (Note, this modification is done by the `murasaki/install` script ).

```
void assert_failed(uint8_t* file, uint32_t line)
{
    CustomAssertFailed(file, line); // debugging stub.
}
```

This hook calls `CustomAssertFailed()` function.

```
// Hook for the assert_failure() in main.c
void CustomAssertFailed(uint8_t* file, uint32_t line)
{
    murasaki::debugger->Printf("Wrong parameters value: file %s on line %d\n", file
    , line);
}
```

Once above programming is done, you can watch the integrity of the HAL parameter by reading the console output.

Above debugging mechanism redirects all HAL assertion, Murasaki assertion and application debug message to the specified logging port. That logging port is able to customize. In this User's Guide, logging is done through the UART port.

Sometimes, you may not want to connect a serial terminal to your system after you saw something wrong. But this is too late because the assertion message is already transmitted ( and lost ).

Murasaki can save this problem. By adding the following code after creating `murasaki::Debugger` instance, you can use history functionality.

```
murasaki::debugger->AutoHistory();
```

The `murasaki::Debugger::AutoHistory()` creates a dedicated task for auto history function. This task watches the input from the logging port. Again, in this User's guide it is UART. Once any character is received from the logging port ( terminal ), previously transmitted message is sent again. Thus you can read the daying message from your application.

The auto history is handy, but it blocks all input from the terminal. If you want to have your own console program through the debug port input, do not use the auto history. Alternatively, you can send the previously transmitted message again, by calling `murasaki::Debugger::PrintHistory()` explicitly.

Murasaki also have post-mortem debugging feature which helps to analyze severe error. Murasaki adds a hook into the `Default_Handler` of the `startup_stm32****.s` file.

```
.section .text.Default_Handler,"ax",%progbits
.global CustomDefaultHandler
Default_Handler:
#if (__ARM_ARCH == 6)
    ldr r0, = CustomDefaultHandler
    bx r0
#else
    b.w CustomDefaultHandler
#endif
Infinite_Loop:
    b Infinite_Loop
```

The inserted instructions supersedes the infinite loop at spurious interrupt handler. Alternatively, `CustomDefaultHandler()` is called. The `CustomDefaultHandler()` stops entire Debugger process, and get into the polling mode serial operation with auto history.

That mean, once spurious interrupt happen, you can read the messages in the debug message FIFO by pressing any key. This feature helps to analyze the assertion message instead of the confusion by unknown trouble.

### 3.4 Tasking

`murasaki::SimpleTask` is a wrapper class of the FreeRTOS task. By using `murasaki::SimpleTask`, a programmer can easily create a task object. This object encapsulates the task of the FreeRTOS.

First of all, you must define a task body function. Any function name is acceptable, Only the return type and parameter type is specified.

```
// Task body of the murasaki::platform.task1
void TaskBodyFunction(const void* ptr)
{
    while (true) // dummy loop
    {
        murasaki::platform.led2->Toggle(); // toggling LED
        murasaki::Sleep(700);
    }
}
```

Then, create a Task object.

There are several parameters to pass for the constructor. The first parameter is the name of the task in FreeRTOS. The second one is the task stack size. The programmer must decide the appropriate size based on the requirement of the task body function. The third one is the priority of the new task. The priority has to be the value of the `murasaki::TaskPriority` type. The fourth one is the pointer to the task parameter. This parameter is passed to the task function body in the run-time. And then, the last one is the pointer to the task body function.

```
// For demonstration of FreeRTOS task.
murasaki::platform.task1 = new murasaki::SimpleTask(
    "Master",
    256,
    murasaki::ktpNormal,
    nullptr,
    &TaskBodyFunction
);
```

Once a task object is created, you must call `Start()` member function to start the task.

```
murasaki::platform.task1->Start();
```

Then, a new task starts.

## 3.5 Other peripherals

This section shows samples of the other peripherals.

- [I2C Master](#)
- [I2C Slave](#)
- [SPI Master](#)
- [SPI Slave](#)
- [GPIO](#)
- [Duplex Audio](#)

### 3.5.1 I2C Master

[murasaki::I2CMaster](#) class provides the serial communication. The I2C master is easy to use. To send a message to the slave device, you need to specify the slave address in 7bits, pointer to data and data size in byte.

```
uint8_t data[5] = { 1, 2, 3, 4, 5 };
murasaki::I2cStatus stat;

stat = murasaki::platform.i2c_master->Transmit(
    127,
    data,
    5);
```

Note: By default, there is no member function "i2c\_master" in the [murasaki::platform](#) variable. The definition and initialization of this member variable are the responsibility of the programmer.

In addition to the [murasaki::I2CMaster::Transmit\(\)](#), [murasaki::I2CMaster](#) class has [murasaki::I2CMaster::Receive\(\)](#), and [murasaki::I2CMaster::TransmitThenReceive\(\)](#) member function.

### 3.5.2 I2C Slave

[murasaki::I2cSlave](#) class provides the I2C slave function. The I2C slave is much easier than master, because it doesn't need to specify the slave address. The I2C slave device address is given by CubeIDE port configuration.

```
uint8_t data[5];
murasaki::I2cStatus stat;

stat = murasaki::platform.i2c_slave->Receive(
    data,
    5);
```

Note: By default, there is no member function "i2c\_slave" in the [murasaki::platform](#) variable. The definition and initialization of this member variable are the responsibility of the programmer.

In addition to the [murasaki::I2cSlave::Transmit\(\)](#), [murasaki::I2cSlave](#) class has [murasaki::I2cSlave::Receive\(\)](#) member function.

### 3.5.3 SPI Master

`murasaki::SpiMaster` is the SPI master class of Murasaki. This class is more complicated than other peripherals because of its flexibility. The SPI master controller must adapt to the several variations of the SPI communication.

- CPOL configuration
- CPHA configuration
- GPIO port configuration to select a slave

To support the above configurations, we need a special mechanism. In Murasaki, this flexibility is the responsibility of the `murasaki::SpiSlaveAdapter` class. This class holds these configurations and passes them to the I2C master class.

So, you must create a `murasaki::SpiSlaveAdapter` class object, at first.

```
// Create a slave adapter. This object specify the protocol and slave select pin
murasaki::SpiSlaveAdapterStrategy * slave_spec;
slave_spec = new murasaki::SpiSlaveAdapter(
    murasaki::kspoFallThenRise,
    murasaki::ksphLatchThenShift,
    SPI_SLAVE_SEL_GPIO_Port,
    SPI_SLAVE_SEL_Pin
);
```

Then, you can pass the `SpiSlaveAdapter` class object to the `murasaki::SpiMaster::TransmitAndReceive()` function.

```
// Transmit and receive data through SPI
uint8_t tx_data[5] = { 1, 2, 3, 4, 5 };
uint8_t rx_data[5];
murasaki::SpiStatus stat;
stat = murasaki::platform.spi_master->
    TransmitAndReceive(
        slave_spec,
        tx_data,
        rx_data,
        5);
```

Note: By default, there are no member function "spi\_master" and "slave\_spec" in the `murasaki::platform` variable. The definition and initialization of these member variables are the responsibility of the programmer.

### 3.5.4 SPI Slave

`murasaki::SpiSlave` class provides the SPI slave functionality. This class encapsulate the SPI slave function.

```
// Transmit and receive data through SPI
uint8_t tx_data[5] = { 1, 2, 3, 4, 5 };
uint8_t rx_data[5];
murasaki::SpiStatus stat;
stat = murasaki::platform.spi_slave->
    TransmitAndReceive(
        tx_data,
        rx_data,
        5);
```

Note: By default, there is no member function "spi\_slave" in the `murasaki::platform` variable. The definition and initialization of this member variable are the responsibility of the programmer.

### 3.5.5 GPIO

`murasaki::BitOut` and `murasaki::BitIn` provides the GPIO functionality. Following is an example of the `murasaki::BitOut` class.

```
// Toggle LED.
murasaki::platform.led->Toggle();
```

Note: By default, there is no member function "led" in the `murasaki::platform` variable. The definition and initialization of this member variable are the responsibility of the programmer.

In addition to the `murasaki::BitOut::Toggle()`, BitOut has `murasaki::BitOut::Set()` and `murasaki::BitOut::Clear()` member function.

### 3.5.6 Duplex Audio

`murasaki::DuplexAudio` class provides a real-time audio IO for both TX and RX together. This class needs a `murasaki::AudioPortAdapterStrategy` object as an interface with hardware.

This class doesn't care about the CODEC IC control. The CODEC initialization and control have to be done by external software.

See the following sample code :

```
// audio CODEC
murasaki::platform.codec = new murasaki::Adau1361(
    48000,           // Fs
    12000000,        // ADAU1361 master clock freq
    murasaki::platform.i2c_master, //
    CODEC port      CODEC_ADDRESS); // I2C device address

murasaki::platform.sai = new murasaki::SaiPortAdaptor(
    &hsai_BlockB1, // TX SAI block
    &hsai_BlockA1); // RX SAI block

murasaki::platform.audio = new murasaki::DuplexAudio(
    murasaki::platform.sai,
    CHANNEL_LEN);
```

The processing of the audio is in the real-time domain. It is recommended to run in the audio task as `murasaki::ktpRealtime` priority. The `murasaki::DuplexAudio::TransmitAndReceive` method is synchronous and blocking. Thus, the processing loop is pretty simple.

```
void TaskBodyFunction(const void* ptr) {
    // audio buffer
    float * l_tx = new float[CHANNEL_LEN];
    float * r_tx = new float[CHANNEL_LEN];
    float * l_rx = new float[CHANNEL_LEN];
    float * r_rx = new float[CHANNEL_LEN];

    murasaki::platform.codec->Start(); // Start the audio codec

    // Loop forever
    while (true) {
        // Talk through
        for (int i = 0; i < CHANNEL_LEN; i++) {
            l_tx[i] = l_rx[i];
            r_tx[i] = r_rx[i];
        }
        murasaki::platform.audio->TransmitAndReceive(l_tx, r_tx, l_rx, r_rx);
    }
}
```



## 3.6 Program flow

In this section, we see the program flow of a Murasaki application. Murasaki has 6 program flows. The start point of these flows are always inside CubeIDE generated code. 3 out of 6 flows are for debugging. Only 1 flow has to be understood well by an application programmer. Rest of 2 are the responsibility of the porting programmer.

- [Application flow](#)
- [HAL Assertion flow](#)
- [Spurious Interrupt flow](#)
- [Assertion flow](#)
- [General Interrupt flow](#)
- [EXTI flow](#)

### 3.6.1 Application flow

The application program flow is the main flow of a Murasaki application. This program flow starts from the [StartDefaultTask\(\)](#) in the [Src/main.c](#). The StartDefaultTask() is default and first task created by CubeIDE. In other words, this task is automatically created without configuration.

From this function, two Murasaki function is called. One is [InitPlatform\(\)](#). The other is [ExecPlatform\(\)](#). Note that both function calls are inserted by murasaki/install script.

```
void StartDefaultTask(void const * argument)
{
    // USER CODE BEGIN 5
    InitPlatform();
    ExecPlatform();
    // Infinite loop
    for(;;)
    {
        osDelay(1);
    }
    // USER CODE END 5
}
```

The [InitPlatform\(\)](#) function is defined in the [Src/murasaki\\_platform.cpp](#). Because the file extension is .cpp, the murasaki\_platform.cpp is compiled by C++ compiler while the [main.c](#) is compiled by C compiler. This allows the programmer to use the C++ language.

As the name suggests, [InitPlatform\(\)](#) is where programmer initialize the platform variables [murasaki::platform](#) and [murasaki::debugger](#).

```
void InitPlatform()
{
    #if ! MURASAKI_CONFIG_NOCYCCNT
        // Start the cycle counter to measure the cycle in MURASAKI_SYSLOG.
        murasaki::InitCycleCounter();
    #endif
    // UART device setting for console interface.
    // On Nucleo, the port connected to the USB port of ST-Link is
    // referred here.
    murasaki::platform.uart_console = new
        murasaki::DebuggerUart(&uart3);
    while (nullptr == murasaki::platform.uart_console)
        ; // stop here on the memory allocation failure.

    // UART is used for logging port.
    // At least one logger is needed to run the debugger class.
    murasaki::platform.logger = new murasaki::UartLogger(
```

```

        murasaki::platform.uart_console);
while (nullptr == murasaki::platform.logger)
    ; // stop here on the memory allocation failure.

// Setting the debugger
murasaki::debugger = new murasaki::Debugger(
    murasaki::platform.logger);
while (nullptr == murasaki::debugger)
    ; // stop here on the memory allocation failure.

// Set the debugger as AutoRePrint mode, for the easy operation.
murasaki::debugger->AutoRePrint(); // type any key to show history.

// For demonstration, one GPIO LED port is reserved.
// The port and pin names are fined by CubeIDE.
murasaki::platform.led = new murasaki::BitOut (LD2_GPIO_Port, LD2_Pin)
;
MURASAKI_ASSERT(nullptr != murasaki::platform.led)

// For demonstration of FreeRTOS task.
murasaki::platform.task1 = new murasaki::SimpleTask(
    "task1",
    256,
    murasaki::ktpNormal,
    nullptr,
    &TaskBodyFunction
);
MURASAKI_ASSERT(nullptr != murasaki::platform.task1)

// Following block is just for sample.
}

```

In this example, the first half of the `InitPlatform()` is building a `murasaki::debugger` variable.

Probably the most critical statement in this part is the creation of the DebuggerUart class object.

```

murasaki::platform.uart_console = new
    murasaki::DebuggerUart(&huart3);

```

In this statement, the DebuggerUart receives the pointer to the huart3 as a parameter. The huart3 is a handle variable of the UART3 generated by CubeIDE. Let's remind the UART3 is utilized as a communication path through the USB in the Nucleo 144 board. So, in this sample code, we are making debugging console through the USB-serial line of the Nucleo F722ZE board.

Because the huart3 is generated into the `main.c` directory, we have to declare this variable as an external variable. You can find the declaration around the top of the `Src/murasaki_platform.cpp`.

```

extern UART_HandleTypeDef huart3;

```

Note that the UART port number depends on the Nucleo board. So, the porting programmer have a responsibility to refer the right UART.

The second half of the `InitPlatform()` is the creative part of the other peripheral object. This part is fully depends on the application. A programmer can define any member variable in the platform variable, by modifying the `murasaki::Platform` struct in the `Inc/platform_defs.hpp`.

The second function which is called from the `StartDefaultTask()` is the `ExecPlatform()`. This function is also defined in the `Src/murasaki_platform.cpp`.

```

void ExecPlatform()
{
    murasaki::platform.task1->Start();

    // print a message with counter value to the console.
    murasaki::debugger->Printf("Push user button to display the I2C slave device \n
        ");

    // Loop forever
    while (true) {
        murasaki::platform.sync_with_button->Wait();
        I2cSearch(murasaki::platform.i2c_master);
    }
}

```

This function is the body of the application. So, you can read GPIO, ADC other peripherals. And output to the DAC, GPIO, and other peripherals from here.

### 3.6.2 HAL Assertion flow

HAL Assertion is a STM32Cube HAL's programming help mechanism.

STM32Cube HAL provides a run-time parameter check. This parameter check is enabled by un-comment the `USE_FULL_ASSERT` macro inside `stm32xxx_hal_conf.h` file. See "Run-time checking" of the HAL manual for detail.

Assertion is defined in `Src/main.c`. As `assert_failed()` function. This function is empty at first. The murasaki install script fills by `CustomAssertFailed()` calling statement.

```
void assert_failed(uint8_t *file, uint32_t line)
{
    // USER CODE BEGIN 6
    CustomAssertFailed(file, line);
    // USER CODE END 6
}
```

If a HAL API received wrong parameter, the `assert_failed()` function is called with its filename and line number. Then, `assert_failed()` call `CustomAssertFailed()` function in the `Src/murasaki_platform.cpp` file.

The `CustomAssertFailed()` prints the filename and line number with the message.

```
void CustomAssertFailed(uint8_t* file, uint32_t line) {
    murasaki::debugger->Printf(
        "Wrong parameters value: file %s on line %d\n",
        file,
        line);
}
```

### 3.6.3 Spurious Interrupt flow

Murasaki provides a mechanism to catch a spurious interrupt. `Default_handler` is the entry point of the spurious interrupt handler. This is defined in `startup/startup_stm32*****.s`.

The install script modify this handler to call the pref `CustomDefaultHandler()` in the `Src/murasaki_platform.cpp`.

```
.section .text.Default_Handler,"ax",%progbits
.global CustomDefaultHandler
Default_Handler:

#if (__ARM_ARCH == 6)
    ldr r0, = CustomDefaultHandler
    bx r0
#else
    b.w CustomDefaultHandler
#endif

Infinite_Loop:
    b Infinite_Loop
```

`CustomDefaultHandler()` is an assembly program. Which pushes register on the stack to allow the `PrintFaultResult` function to print out the register and exception environment. After printing, the system gets into the post-mortem mode which responses any key from the console and then flush out the contents of printf message FIFO.

Note that the `CustomDefaultHandler()` works correctly only when both conditions are met :

- Core is ARM v7m ( The CORTEX-Mx except for M0, M0+ )
- Murasaki is the release build.
- The exception is Hard Fault.

The Debug build makes an unexpected stack frame in the entry code of the `HardFaultHandler`. Thus, the printed resources by the debug build are not precise.

### 3.6.4 Assertion flow

The assertion flow is similar to the Spurious Interrupt flow. Once the assertion is raised, the assertion macro raises Hard Fault exception. The Hard Fault exception handler in the Src/st32\*\*\*\_it.c calls CustomDefaultHandler.

```
void HardFault_Handler(void)
{
    CustomDefaultHandler();
    while (1)
    {
    }
}
```

### 3.6.5 General Interrupt flow

As described in the HAL manual, STM32Cube HAL handles all interrupts relevant to the peripherals, and then, call the corresponding callback function. These callbacks are optional from the viewpoint of the peripheral hardware, but an essential hook to sync with the software.

Murasaki is using these callbacks to notify the end of processing, to the peripheral class objects. These callbacks are managed automatically. For example, the following is the UART RX completion call back inside [murasaki\\_callback.cpp](#).

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    // Obtain the responding object.
    murasaki::PeripheralStrategy *peripheral =
        murasaki::CallbackRepositorySingleton::GetInstance()->
        GetPeripheralObject(huart);
    // Convert it to the appropriate type.
    murasaki::UartStrategy *uart = reinterpret_cast<
        murasaki::UartStrategy*>(peripheral);
    // Handle the callback by the object.
    uart->ReceiveCompleteCallback(huart);
}
```

This callback is called from HAL, after the end of peripheral interrupt processing. Then, the [murasaki::CallbackRepositorySingleton](#) class search for a class object which has responsibility to the given peripheral handle.

Then, its appropriate member function is called.

### 3.6.6 EXTI flow

The EXTI flow is very similar to the [General Interrupt flow](#) except its timing.

By design, CubeIDE generated code enable the EXTI right after main calls MX\_GPIO\_Init(). This means EXTIs are accepted even before Murasaki prepare the interrupt routines. If EXTI is accepted before Murasaki is ready, it will cause assertion failure. To avoid it, Murasaki disable the interrupt callback handling in the [HAL\\_GPIO\\_EXTI\\_Callback\(\)](#) until first [murasaki::Exti::Wait\(\)](#) is executed.

So, all EXTI interrupt before the first Exti::Wait() call is ignored.

## 4 Porting guide

This porting guide introduces murasaki class library porting. In this guide, user will study the library porting to the STM32 microcomputer system working with STM32Cube HAL. A step by step procedure with screen capture is explained in [a separated document](#).

Followings are the contents of this porting guide :

- [Directory Structure](#)
- [CubeIDE setting](#)
- [Configuration](#)
- [Task Priority and Stack Size](#)
- [Heap memory consideration](#)
- [Platform variable](#)
- [Routing interrupts](#)
- [Error handling](#)
- [Summary of the porting](#)

There are some other manuals of murasaki class library :

- [Preface](#)
- [Usage Introduction](#)
- [Murasaki Class Collection](#)

### 4.1 Directory Structure

Murasaki has four main directory and several user-modifiable files. This page describes these directories and files.

#### 4.1.1 Src directory

Almost files of the Murasaki source code are stored in this directory. Basically, there is no need to edit the files inside this directory, except the development of Murasaki itself. The CubeIDE project setting must refer this directory as the source directory.

#### 4.1.2 Inc directory

This directory contains the include files, the CubeIDE project setting must refer this directory as an include directory.

#### 4.1.3 Src/Thirdparty and Inc/Thirdparty directory

The class collection of the third party peripherals. The "third party" means outside of the microprocessor.

#### 4.1.4 `murasaki.hpp`

Usually, the `murasaki.hpp` include file is the only one to include from an application program. By including this file, an application can refer all the definition of the Murasaki

This file is stored in the `Inc` directory.

#### 4.1.5 `template directory`

##### 4.1.5.1 `platform_config.hpp`

The `platform_config.hpp` file is a collection of the build configuration. By defining a macro, a programmer can change the behavior of the Murasaki. There are mainly two types of the configuration in this file.

One type of configuration is to override the `murasaki_config.hpp` file. All contents of the `murasaki_config.hpp` are macros. These macros are defined to control the Murasaki, for example: the task priority, the task stack size or the timeout period. Refer [Definitions and Configuration](#).

The other configuration type is the assertion inside Murasaki. See `MURASAKI_CONFIG_NODEBUG` for details.

The `platform_config.hpp` is better to be copied in the `/Inc` directory of the application. The `install script` will copy this file to `/Src` directory of application.

##### 4.1.5.2 `platform_defs.hpp`

As same as `platform_config.hpp`, the `platform_defs.hpp` is not the core part of the Murasaki class library. This include file has a definition of the `murasaki::platform` variable which provides "nice looking" aggregation of the class objects, rather than spreading them over the global scope.

The application programmer can define the `murasaki::Platform` type freely. There is no limitation or requirement what you put into unless compiler reports an error message.

On the other hand, a programmer may find that adding the peripheral-based class variables and middleware based class variables into the `murasaki::Platform` type is reasonable. Actually, the independent devices ( ie:I2C connected LCD controller ) may be better to be a member variable of the `mruasaki::Platform` type, just because it is easy to access.

The `platform_defs.hpp` is better to be copied in the `/Inc` directory of the application. The `install script` will copy this file to `/Src` directory of application for programmer.

See [Application Specific Platform](#) as usage sample.

##### 4.1.5.3 `murasaki_platform.hpp`

A header file of the `murasaki_platform.cpp`. This file is better to be copied in the `/Inc` directory of the application. The `install script` will copy this file to `/Src` directory of application.

#### 4.1.5.4 murasaki\_platform.cpp

The [murasaki\\_platform.cpp](#) is the interface between the application and the HAL/RTOS. This file has variables / functions which user needs to program at porting time.

- [murasaki::platform](#) variable
- [murasaki::debugger](#) variable
- [InitPlatform\(\)](#) to initialize the platform variable
- [ExecPlatform\(\)](#) to execute the platform algorithm
- [Interrupt routing functions](#)
- [HAL assertion function and Custom default exception handler](#)

The [murasaki\\_platform.cpp](#) is better to be copied in the /Src directory of the application. The [install script](#) will copy this file to /Src directory of application.

#### 4.1.6 install script

The install script have mainly 4 tasks.

- Copy template files to the appropriate application directories from [template directory](#)
- Modify [main.c](#) to call the [InitPlatform\(\)](#) and [ExecPlatform\(\)](#) from the default task.
- Modify [main.c](#) to call the [CustomAssertFailed\(\)](#) from the HAL assertion
- Modify the hard fault handler to call the [CustomDefaultHandler\(\)](#)
- Generate [murasaki\\_include\\_stub.h](#) to let the Murasaki library to include HAL headers.

Last one is little bit tricky to do it manually. Refer [murasaki\\_include\\_stub.h](#) for details.

## 4.2 CubeIDE setting

There is several required CubeIDE setting.

- [Heap Size](#)
- [Stack Size](#)
- [Task stack size of the default task](#)
- [UART peripheral](#)
- [SPI Master peripheral](#)
- [SPI Slave peripheral](#)
- [I2C peripheral](#)
- [EXTI](#)

#### 4.2.1 Heap Size

Heap is very important in the application with murasaki.

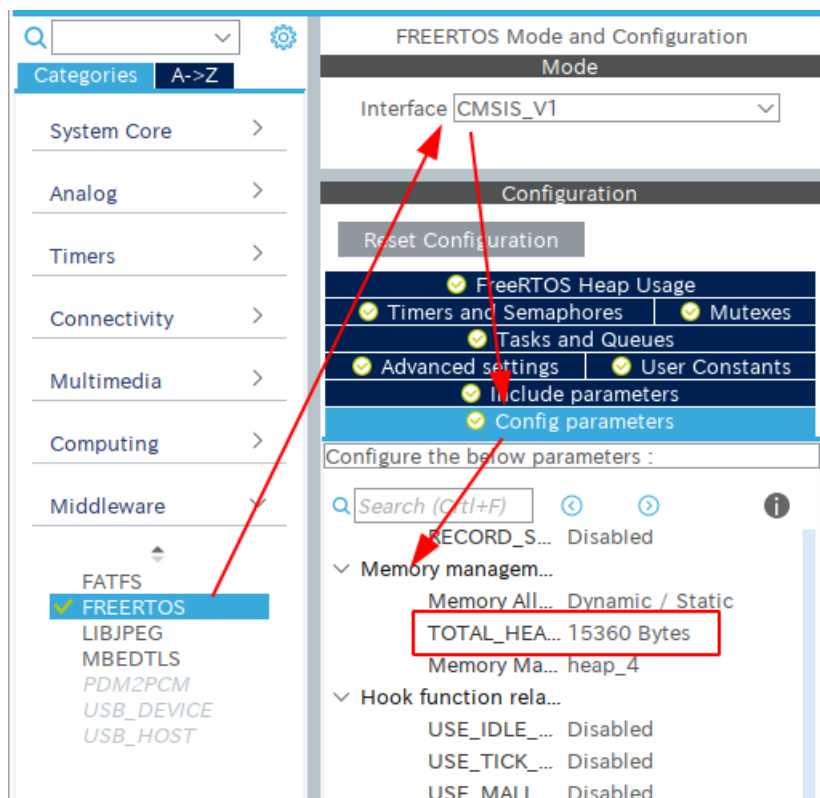
First, class instances are created inside heap region by new operator often. And second, `murasaki::Debugger` allocates a huge size of FIFO buffer. This buffer stays in between the `murasaki::Debugger::Printf()` function and the logger task. The size of this FIFO buffer is defined by `PLATFORM_CONFIG_DEBUG_BUFFER_SIZE`. The default is 4KB.

Usually, the heap is simply called "heap", without precise definition of terminology. But let's call it "system heap" here. The system heap is the one which is managed by new and delete operators by default.

In addition to the system heap, FreeRTOS has its own heap. This heap is managed separately from the system heap. The management contains the heap size watching and returning error. And this heap is thread safe while the system heap is not.

Using two heap is not easy. And definitely, the FreeRTOS heap is better than the system heap in the embedded application. So, in murasaki, the new and the delete operators are overloaded and redirected to the FreeRTOS heap. See [Heap memory consideration](#) for detail.

To avoid the heap allocation problem, it is better to have more than 16kB FreeRTOS heap. The FreeRTOS heap size can be changed by CubeIDE.



On the other hand, the system heap size can be smaller like 128 Byte because we don't use it..

Note that to know the minimum requirement of the system heap size, you must investigate how much allocations are done before entering FreeRTOS. Because murasaki application doesn't use any system heap, only very small management memory should be required in system heap.

The system Heap size can be set by following place.



Pinout & Configuration	Clock Configuration	Project Manager
<div>Project</div> <div>Code Generator</div> <div>Advanced Settings</div>	<div>Project Settings</div> <div>Project Name</div> <div>nucleo-f446-64-release-preparation</div> <div>Project Location</div> <div>/home/takemasa/git/murasaki_dev</div> <div>Application Structure</div> <div>Advanced</div> <div><input type="checkbox"/> Do not generate the main()</div> <div>Toolchain Folder Location</div> <div>/home/takemasa/git/murasaki_dev/nucleo-f446-64-release-preparation/</div> <div>Toolchain / IDE</div> <div>STM32CubeIDE</div> <div><input checked="" type="checkbox"/> Generate Under Root</div> <div>Linker Settings</div> <div>Minimum Heap Size</div> <div>0x200</div> <div>Minimum Stack Size</div> <div>0x400</div>	

#### 4.2.2 Stack Size

In this section, the stack means the interrupt stack.

The interrupt stack is used only when the interrupt is accepted. Then, it is basically small.

By the way, murasaki uses its assertion often. Once assertion fails, a message is created by `snprintf()` function and transmitted through FIFO. These operations consume stack. And assertion can be happen also in the ISR context.

The debugging in the ISR is not easy without assertion and `printf()`. To make them always possible, it is better to set the interrupt stack size bigger than 256 Bytes. The interrupt stack size can be changed by CubeIDE :

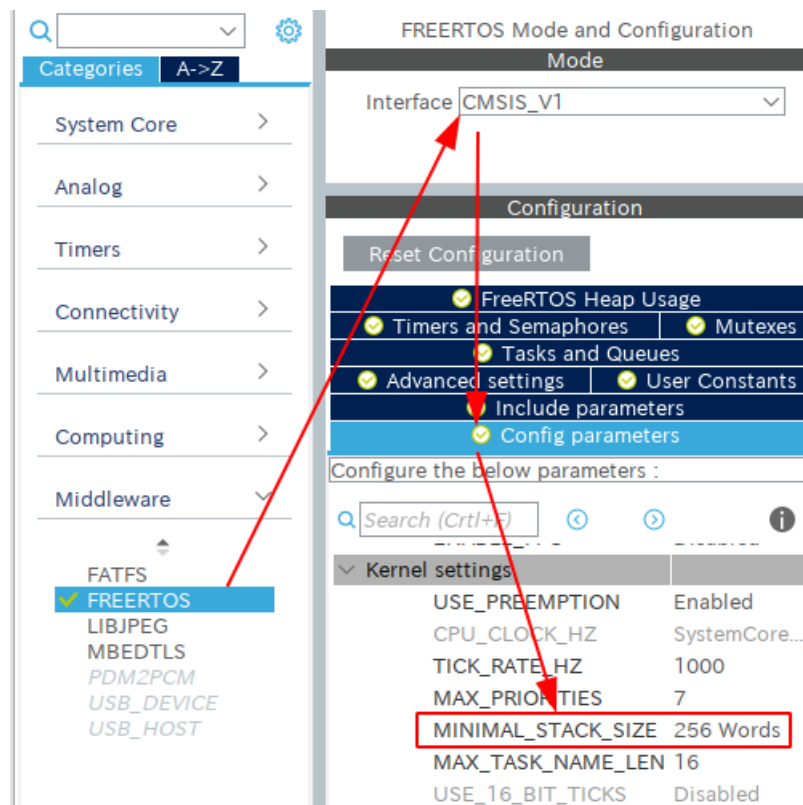
Pinout & Configuration	Clock Configuration	Project Manager
<div>Project</div> <div>Code Generator</div> <div>Advanced Settings</div>	<div>Project Settings</div> <div>Project Name</div> <div>nucleo-f446-64-release-preparation</div> <div>Project Location</div> <div>/home/takemasa/git/murasaki_dev</div> <div>Application Structure</div> <div>Advanced</div> <div><input type="checkbox"/> Do not generate the main()</div> <div>Toolchain Folder Location</div> <div>/home/takemasa/git/murasaki_dev/nucleo-f446-64-release-preparation/</div> <div>Toolchain / IDE</div> <div>STM32CubeIDE</div> <div><input checked="" type="checkbox"/> Generate Under Root</div> <div>Linker Settings</div> <div>Minimum Heap Size</div> <div>0x200</div> <div>Minimum Stack Size</div> <div>0x400</div>	

#### 4.2.3 Task stack size of the default task

The default task stack size is very small( 128 Bytes )

This is not enough to use murasaki and its debugger output functionality. It should be increased at smallest 256 Bytes.

It can be changed by CubeIDE:



#### 4.2.4 UART peripheral

UART/USART peripheral have to be configured as Asynchronous mode. The DMA have to be enabled for both TX and RX. Both DMA must be normal mode. All the NVIC interrupts have to be enabled. See [murasaki::Uart](#) for details.

#### 4.2.5 SPI Master peripheral

SPI Master peripheral have to be configured as Full-Duplex Master mode. The NSS must be disabled. The DMA have to be enabled for both TX and RX. Both DMA must be normal mode. All the NVIC interrupt have to be enabled. See [murasaki::SpiMaster](#) for details.

#### 4.2.6 SPI Slave peripheral

SPI Slave peripheral have to be configured as Full-Duplex Slave mode. The NSS must be input signal. The DMA have to be enabled for both TX and RX. Both DMA must be normal mode. All the NVIC interrupt have to be enabled. See [murasaki::SpiSlave](#) for details.

#### 4.2.7 I2C peripheral

I2C have to be configured as "I2" mode. To configure as I2C device, the primary slave address have to be configured. The NVIC interrupt have to be enabled. See [murasaki::I2cMaster](#) for details.

#### 4.2.8 EXTI

The corresponding interrupt have to be enabled by NVIC. See [murasaki::Exti](#) for details.

### 4.3 Configuration

Murasaki has configurable parameters. These parameters control mainly the task size and task priority. One of the special configurations is [MURASAKI\\_CONFIG\\_NODEBUG](#) macro. This macro controls whether assertion inside Murasaki source code works or ignored.

To customize the configuration, define the configuration macro with the desired value in the [platform\\_config.hpp](#) file. This definition will override the Murasaki default configuration.

For the detail of each macro, see [Definitions and Configuration](#).

### 4.4 Task Priority and Stack Size

The Murasaki task priority is from [murasaki::ktpIdle](#) to [murasaki::ktpRealtime](#).

At the initial state, the Murasaki has two hidden tasks inside. Both are running for the [murasaki::Debugger](#) class, and both task's priority are defined as [PLATFORM\\_CONFIG\\_DEBUG\\_TASK\\_PRIORITY](#). By default, the value of [PLATFORM\\_CONFIG\\_DEBUG\\_TASK\\_PRIORITY](#) is [murasaki::ktpHigh](#). That means, debug tasks priority is very high.

The debug tasks should have priority as high as possible. Otherwise, another task may block the debugging message.

Unlike the task priority, the interrupt priority is easy. Usually, it is not so sensitive because the ISR is very short in the good designed RTOS application design. In this case, all ISR can be a same priority.

In the bad designed RTOS application, there are very few things we can do. Such the things are project dependent.

### 4.5 Heap memory consideration

In Murasaki, there is a re-definition of [operator new](#) and [operator delete](#) inside [allocators.cpp](#). This re-definition let the [pvPortMalloc\(\)](#) allocate a fragment of memory for the [operator new](#).

These changes converge all memory allocation to the FreeRTOS's heap. There is some advantage of this convergence:

- The FreeRTOS heap is thread safe while the system heap in CubeIDE is not thread-safe
- The FreeRTOS heap is checking the heap size limitation and return an error, while the system heap behavior in CubeIDE is not clear.
- The heap size calculation is easier if we integrate the memory allocation activity into one heap.

On the other hand, FreeRTOS heap is not able to allocate/deallocate in the ISR context. And it is impossible to use the FreeRTOS heap before starting up the FreeRTOS. Then, we have to follow the rules here :

- C++ new / delete operators have to be called after FreeRTOS started.
- C++ new / delete operators have to be called in the task context.

## 4.6 Platform variable

The `murasaki::platform` and the `murasaki::debugger` have to be initialized by the `InitPlatform()` function. The programming of this function is a responsibility of the porting programmer.

The porting programmer has to make the peripheral handles as visible from the `murasaki_platform.cpp`. For example, CubeMx generate the huart2 for Nucleo L152RE for the serial communication over the ST-LINK USB connection. huart2 is defined in `main.c` as like below:

```
UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart2_rx;
DMA_HandleTypeDef hdma_usart2_tx;
```

To use this handle, the porting programmer has to declare the same name as an external variable, in the `murasaki_platform.cpp` :

```
extern UART_HandleTypeDef huart2;
```

After these preparations, the porting programmer can program the `InitPlatform()` :

```
void InitPlatform()
{
    // UART device setting for console interface.
    // On Nucleo, the port connected to the USB port of ST-Link is
    // referred here.
    murasaki::platform.uart_console = new
        murasaki::DebuggerUart(&huart2);
    while (nullptr == murasaki::platform.uart_console)
        ; // stop here on the memory allocation failure.

    // UART is used for logging port.
    // At least one logger is needed to run the debugger class.
    murasaki::platform.logger = new murasaki::UartLogger(
        murasaki::platform.uart_console);
    while (nullptr == murasaki::platform.logger)
        ; // stop here on the memory allocation failure.

    // Setting the debugger
    murasaki::debugger = new murasaki::Debugger(
        murasaki::platform.logger);
    while (nullptr == murasaki::debugger)
        ; // stop here on the memory allocation failure.

    // Set the debugger as AutoRePrint mode, for the easy operation.
    murasaki::debugger->AutoRePrint(); // type any key to show history.

    // For demonstration, one GPIO LED port is reserved.
    // The port and pin names are fined by CubeIDE.
    murasaki::platform.led = new murasaki::BitOut(LD2_GPIO_Port, LD2_Pin)
        ;
    MURASAKI_ASSERT(nullptr != murasaki::platform.led)
}
```

In this example, we initialize the `uart_console` member variable which is `murasaki::UartStrategy` class. The application programmer control the UART2 through this `uart_console` member variable.

In the second step, we pass this `uart_console` to the `logger` member variable. This member variable is an essential stub for the `murasaki::debugger`. In this example, we assign the UART2 port as interface for the debugging output.

After the logger becomes ready, we initialize the `murasaki::debugger`. As we already discussed, this debugger receives a logger object as a parameter. The debugger output all messages through this logger.

The last step is optional. We invoke the `murasaki::Debugger::AutoRePrint()` member function. By calling this function, logger re-print the old data in the FIFO again whenever the end-user type any key of the keyboard.

This "auto re-print by any key" is convenient in the small system. But for the large system which has its own command line shell, this input-interruption is harmful. For such the system, programmer want to call `murasaki::Debugger::RePrint()` member function, by certain customer command.

Once the debugger is ready to use, we create the led member variable as a general purpose output port of the application .

The `ExecPlatform()` function implements the actual algorithm of application. In the example below, the application is blinking a LED and printing a messages on the console output.

```
void ExecPlatform()
{
    // counter for the demonstration.
    int count = 0;

    // Loop forever
    while (true) {
        // Toggle LED.
        murasaki::platform.led->Toggle();

        // print a message with counter value to the console.
        murasaki::debugger->Printf("Hello %d \n", count);

        // update the counter value.
        count++;

        // wait for a while
        murasaki::Sleep(500);
    }
}
```

Finally, above two functions have to be called from `StartDefaultTask` of the `main.c`. Also, `main.c` must include the `murasaki_platform.hpp` to read the prototype of these functions.

Following is the example of the `StartDefaultTask()`. The actual code have a comment to work together the code generator of the CubeIDE. But this sample remove them because of the documentation tool ( doxygen ) limitation.

```
void StartDefaultTask(void const * argument)
{
    InitPlatform();
    ExecPlatform();

    for(;;)
    {
        osDelay(1);
    }
}
```

## 4.7 Routing interrupts

The `murasaki_platform.cpp` has skeletons of HAL callback. These callbacks are pre-defined inside HAL as receptors of interrupt. These definitions inside HAL are "weak" binding. Thus, these skeletons in `murasaki_platform.cpp` overrides the definition. The porting programmer have to program these skeletons correctly.

Murasaki is using these callback to notify the end of processing, to the peripheral class objects. These callback are managed automatically. For example, the following is the UART RX completion call back inside `murasaki_callback.cpp`.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    // Obtain the responding object.
    murasaki::PeripheralStrategy *peripheral =
        murasaki::CallbackRepositorySingleton::GetInstance()->
        GetPeripheralObject(huart);
    // Convert it to the appropriate type.
    murasaki::UartStrategy *uart = reinterpret_cast<
        murasaki::UartStrategy*>(peripheral);
    // Handle the callback by the object.
    uart->ReceiveCompleteCallback(huart);
}
```

This callback is called from HAL, after the end of peripheral interrupt processing. Then, the [murasaki::CallbackRepositorySingleton](#) class search for a class object which has responsibility to the given peripheral handle.

Then, its appropriate member function is called.

Following is the list of the interrupts which applicaiton have to route to the peripheral class variables.

- void [HAL\\_UART\\_TxCpltCallback](#)(UART\_HandleTypeDef \*huart);
- void [HAL\\_UART\\_RxCpltCallback](#)(UART\_HandleTypeDef \*huart);
- void [HAL\\_UART\\_ErrorCallback](#)(UART\_HandleTypeDef \*huart);
- void [HAL\\_SPI\\_TxRxCpltCallback](#)(SPI\_HandleTypeDef \*hspi);
- void [HAL\\_SPI\\_ErrorCallback](#)(SPI\_HandleTypeDef \*hspi);
- void [HAL\\_I2C\\_MasterTxCpltCallback](#)(I2C\_HandleTypeDef \*hi2c);
- void [HAL\\_I2C\\_MasterRxCpltCallback](#)(I2C\_HandleTypeDef \*hi2c);
- void [HAL\\_I2C\\_SlaveTxCpltCallback](#)(I2C\_HandleTypeDef \*hi2c);
- void [HAL\\_I2C\\_SlaveRxCpltCallback](#)(I2C\_HandleTypeDef \*hi2c);
- void [HAL\\_I2C\\_ErrorCallback](#)(I2C\_HandleTypeDef \*hi2c);
- void [HAL\\_SAI\\_RxHalfCpltCallback](#)(SAI\_HandleTypeDef \*hsai);
- void [HAL\\_SAI\\_RxCpltCallback](#)(SAI\_HandleTypeDef \*hsai);
- void [HAL\\_SAI\\_ErrorCallback](#)(SAI\_HandleTypeDef \*hsai) ;
- void [HAL\\_I2S\\_RxHalfCpltCallback](#)(SAI\_HandleTypeDef \*hsai);
- void [HAL\\_I2S\\_RxCpltCallback](#)(SAI\_HandleTypeDef \*hsai);
- void [HAL\\_I2S\\_ErrorCallback](#)(SAI\_HandleTypeDef \*hsai) ;
- void [HAL\\_ADC\\_ConvCpltCallback](#)(ADC\_HandleTypeDef \*hadc);
- void [HAL\\_ADC\\_ErrorCallback](#)(ADC\_HandleTypeDef \*hadc);
- void [HAL\\_GPIO\\_EXTI\\_Callback](#)(uint16\_t GPIO\_P);

## 4.8 Error handling

The [murasaki\\_platform.cpp](#) has two error handling functions. These functions are pre-programmed from the first. And usually its enough to use the pre-programmed version. On the other hand the porting programmer have to modify the application program to call these error handling functions at appropriate situation. Otherwise, these error handling functions will be never called.

The [CustomAssertFailed\(\)](#) function should be called from the [assert\\_failed\(\)](#) function. The [assert\\_failed\(\)](#) function is located in the [main.c](#). Modifying the [assert\\_failed\(\)](#) is the responsibility of the porting programmer.

```
void assert_failed(uint8_t* file, uint32_t line)
{
    CustomAssertFailed(file, line);
}
```

To enable the [assert\\_failed\(\)](#), the porting programmer have to enable the HAL internal assertion from the CubeIDE.

The [CustomDefaultHandler\(\)](#) function should be called from the default exception routine. But the system default exception handler ( `Default_Handler` ) doesn't do anything by default. To maximize the information to the JTAG debugger, this is programmed as very simple eternal loop.

The default exception handler can be programmed or left untouched as porting programmer want. It is up to the system policy. If it is re-programmed to call the [CustomDefaultHandler\(\)](#), [murasaki::debugger](#) object take the control of the debug message FIFO at the exception handler context.

If the exception happened and the `CustomDefaultHandler` is called, the end user can see the entire messages in the debug FIFO by typing any key of the keyboard. This is useful to see the last message from the assertion. The last message usually represent the cause of the exception. The end user can debug the application program based on this last assertion message.

The HAL default exception routine is programmed at startup/startup\_stm32xxxx.s by assembly language.

The porting programmer can modify it as below, to call the [CustomDefaultHandler\(\)](#);

```
.section .text.Default_Handler,"ax",%progbits
.global CustomDefaultHandler
Default_Handler:
#if (__ARM_ARCH == 6)
    ldr r0, = CustomDefaultHandler
    bx r0
#else
    b.w CustomDefaultHandler
#endif
Infinite_Loop:
    b Infinite_Loop
```

## 4.9 Summary of the porting

Following is the porting steps :

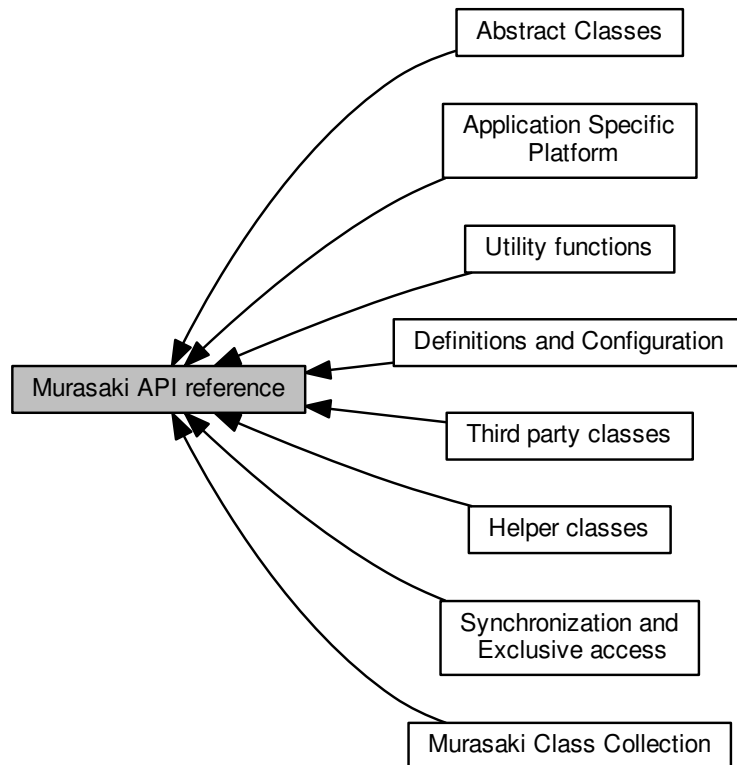
- Adjust heap size and stack size as described in the [CubeIDE setting](#)
- Generate an application skeleton from CubeIDE.
- Checkout Murasaki repository into your project.
- Copy the template files as described in the [Directory Structure](#) .
- Configure Muraaski as described in the [Configuration](#) and the [Task Priority and Stack Size](#)
- Call [InitPlatform\(\)](#) and [ExecPlatform\(\)](#) as described [Platform variable](#).
- Route the interrupts as described [Routing interrupts](#).
- Route the error handling as described [Error handling](#)

## 5 Module Documentation

### 5.1 Murasaki API reference

Murasaki API reference place holder.

Collaboration diagram for Murasaki API reference:



#### Modules

- [Murasaki Class Collection](#)  
*STM32 Class library.*
- [Third party classes](#)  
*Classes for the thirdparty devices.*
- [Definitions and Configuration](#)  
*Definitions and configuration collection of murasaki platform.*
- [Application Specific Platform](#)  
*Variables to control the hardware.*
- [Abstract Classes](#)  
*Generic classes as template of the concrete class.*
- [Synchronization and Exclusive access](#)  
*Sync between the task and interrupt. Make the resources thread safe.*



- [Helper classes](#)

*Classes to support the murasaki-class.*

- [Utility functions](#)

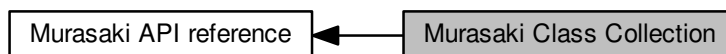
*Collection of the useful functions.*

### 5.1.1 Detailed Description

## 5.2 Murasaki Class Collection

STM32 Class library.

Collaboration diagram for Murasaki Class Collection:



### Classes

- class [murasaki::Adc](#)  
*STM32 dedicated ADC class.*
- class [murasaki::BitIn](#)  
*General purpose bit input.*
- class [murasaki::BitOut](#)  
*General purpose bit output.*
- class [murasaki::Debugger](#)  
*Debug class. Provides printf() style output for both task and ISR context.*
- class [murasaki::DuplexAudio](#)  
*Stereo Audio is served by this class.*
- class [murasaki::Exti](#)  
*EXTI wrapper class.*
- class [murasaki::I2cMaster](#)  
*Thread safe, synchronous, and blocking IO. Encapsulating I2C master. Based on STM32Cube HAL driver and FreeRTOS.*
- class [murasaki::I2cSlave](#)  
*Thread safe, synchronous and blocking IO. Encapsulating I2C slave. Based on STM32Cube HAL driver and FreeRTOS.*
- class [murasaki::I2sPortAdapter](#)  
*Adapter as I2S audio port.*
- class [murasaki::QuadratureEncoder](#)  
*Quadrature Encoder class.*
- class [murasaki::SaiPortAdapter](#)  
*Adapter as SAI audio port.*
- class [murasaki::SimpleTask](#)  
*An easy to use task class.*
- class [murasaki::SpiMaster](#)  
*Thread safe, synchronous and blocking IO. Encapsulating SPI master. Based on STM32Cube HAL driver and FreeRTOS.*
- class [murasaki::SpiSlave](#)  
*Thread safe, synchronous and blocking IO. Encapsulating SPI slave. Based on STM32Cube HAL driver and FreeRTOS.*
- class [murasaki::SpiSlaveAdapter](#)  
*A specifier of SPI slave.*
- class [murasaki::Uart](#)  
*Thread safe, synchronous and blocking IO. Concrete implementation of UART controller. Based on the STM32Cube HAL DMA Transfer.*
- class [murasaki::UartLogger](#)  
*Logging through an UART port.*

## Macros

- `#define MURASAKI_ASSERT(COND)`  
*Assert the COND is true.*
- `#define MURASAKI_PRINT_ERROR(ERR)`  
*Print ERR if ERR is true.*
- `#define MURASAKI_SYSLOG(OBJPTR, FACILITY, SEVERITY, FORMAT, ...)`  
*output The debug message*

### 5.2.1 Detailed Description

This page is a reference guide to the murasaki class library. This guide describes class by class and cover the entire library. It is not recommended to read the reference for the first time user.

A programmer should read the [Usage Introduction](#) as the first step.

### 5.2.2 Macro Definition Documentation

#### 5.2.2.1 `#define MURASAKI_ASSERT( COND )`

##### Value:

```
if ( ! (COND) )\
{
    murasaki::debugger->Printf("-----\n");
    murasaki::debugger->Printf(MURASAKI_ASSERT_MSG, __func__, __LINE__,
    ,__MURASAKI_FILE__ );\
    murasaki::debugger->Printf("Fail expression : %s\n", #COND);\
    { void (*foo)(void) = (void (*)())1; foo(); }\
}
```

##### Parameters

<code>COND</code>	Condition as bool type.
-------------------	-------------------------

Print the COND expression to the logging port if COND is false. Do nothing if CODN is true.

After printing the assertion failure message, this aspersion triggers the Hard Fault exception. The Hard Fault Exception is caught by [HardFault\\_Handler\(\)](#) and eventually invoke the `murasaki::debugger->DoPostMortem()`, to put the system into the post mortem debug mode.

The following code in the macro definition calls a non-existing function located address 1. Such access causes a hard fault exception.

```
1 { void (*foo)(void) = (void (*)())1; foo(); }\
```

This assertion does nothing if the programmer defines `MURASAKI_CONFIG_NODEBUG` macro as true. This macro is defined in the file [platform\\_config.hpp](#).

### 5.2.2.2 #define MURASAKI\_PRINT\_ERROR( ERR )

#### Value:

```
if ( (ERR) ) \
{ \
    murasaki::debugger->Printf(MURASAKI_ERROR_MSG, __func__, __LINE__,
    __MURASAKI__FILE__, #ERR ); \
}
```

#### Parameters

<b>ERR</b>	Condition as bool type.
------------	-------------------------

Print the ERR expression to the logging port if COND is true. Do nothing if ERR is true.

This assertion does nothing if the programmer defines [MURASAKI\\_CONFIG\\_NODEBUG](#) macro as true. This macro is defined in the file [platform\\_config.hpp](#).

For example, the following code is a typical usage of this macro. ERROR macro is copied from STM32Cube HAL source code.

```
1 bool Uart::HandleError(void* const ptr)
2 {
3     MURASAKI_ASSERT(nullptr != ptr)
4
5     if (peripheral_ == ptr) {
6         // Check error, and print if exist.
7         MURASAKI_PRINT_ERROR(peripheral_>ErrorCode & HAL_UART_ERROR_DMA);
8         MURASAKI_PRINT_ERROR(peripheral_>ErrorCode & HAL_UART_ERROR_PE);
9         MURASAKI_PRINT_ERROR(peripheral_>ErrorCode & HAL_UART_ERROR_NE);
10        MURASAKI_PRINT_ERROR(peripheral_>ErrorCode & HAL_UART_ERROR_FE);
11        MURASAKI_PRINT_ERROR(peripheral_>ErrorCode & HAL_UART_ERROR_ORE);
12        MURASAKI_PRINT_ERROR(peripheral_>ErrorCode & HAL_UART_ERROR_DMA);
13        return true;    // report the ptr matched
14    }
15    else {
16        return false;    // report the ptr doesn't match
17    }
18 }
```

### 5.2.2.3 #define MURASAKI\_SYSLOG( OBJPTR, FACILITY, SEVERITY, FORMAT, ... )

#### Parameters

<b>OBJPTR</b>	the pointer to the object. Usually, path the "this" pointer here.
<b>FACILITY</b>	Specify which facility makes this log. Choose from <a href="#">murasaki::SyslogFacility</a>
<b>SEVERITY</b>	Specify how message is severe. Choose from <a href="#">murasaki::SyslogSeverity</a>
<b>FORMAT</b>	Message format as printf style.

Output the debugg message to debug console output.

The output message is filtered by the internal threshold set by [murasaki::SetSyslogSererityThreshold](#), [murasaki::↵SetSyslogFacilityMask](#) and [murasaki::AddSyslogFacilityToMask](#). See these function's document to understand how filter works.

There is recommendation in the SEVERITY parameter :

- [murasaki::kseDebug](#) for Development/Debug message for tracing normal operation.
- [murasaki::kseWarning](#) for relatively severe condition which need abnormal action, or cannot handle.
- [murasaki::kseError](#) for faulty condition from HAL or hardware.
- [murasaki::kseEmergency](#) for software logic error like assert fail

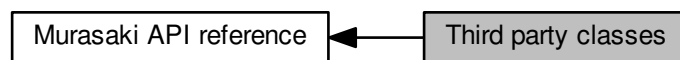
The output format is as following :

- Clock cycles by [GetCycleCounter\(\)](#)
- Object address
- Facility
- Severity
- File name of source code
- Line number of source code
- Function name
- Other programmer specified information

### 5.3 Third party classes

Classes for the thirdparty devices.

Collaboration diagram for Third party classes:



#### Classes

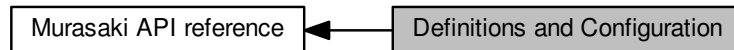
- class [murasaki::Adau1361](#)  
*Audio Codec LSI class.*

#### 5.3.1 Detailed Description

## 5.4 Definitions and Configuration

Definitions and configuration collection of murasaki platform.

Collaboration diagram for Definitions and Configuration:



- `#define PLATFORM_CONFIG_DEBUG_LINE_SIZE 256`  
*Size of one line[byte] in the debug printf.*
- `#define PLATFORM_CONFIG_DEBUG_BUFFER_SIZE 4096`  
*Size[byte] of the circular buffer to be transmitted through the serial port.*
- `#define PLATFORM_CONFIG_DEBUG_SERIAL_TIMEOUT (murasaki::kwmsIndefinitely)`  
*Timeout of the serial port to transmit the string through the Debug class.*
- `#define PLATFORM_CONFIG_DEBUG_TASK_STACK_SIZE 256`  
*Size[Byte] of the task inside Debug class.*
- `#define PLATFORM_CONFIG_DEBUG_TASK_PRIORITY murasaki::ktpHigh`  
*The task priority of the debug task.*
- `#define MURASAKI_CONFIG_NODEBUG false`  
*Suppress `MURASAKI_ASSERT` macro.*
- `#define MURASAKI_CONFIG_NOCYCCNT false`  
*Doesn't run the CYCCNT counter.*
- `#define NUM_OF_CALLBACK_OBJECTS 8`  
*The number of the interrupt callback handling objects.*
- `#define NUM_OF_EXTI_OBJECTS 8`  
*The number of the EXTI callback handling objects.*

### 5.4.1 Detailed Description

### 5.4.2 Macro Definition Documentation

#### 5.4.2.1 `#define MURASAKI_CONFIG_NOCYCCNT false`

Set this macro to true, to halt the CYCCNT counter. Set this macro false, to run.

To override the definition here, define same macro inside [platform\\_config.hpp](#).

#### 5.4.2.2 `#define MURASAKI_CONFIG_NODEBUG false`

Set this macro to true, to discard the assertion `MURASAKI_ASSERT`. Set this macro false, to use the assertion.

To override the definition here, define same macro inside [platform\\_config.hpp](#).

#### 5.4.2.3 `#define NUM_OF_CALLBACK_OBJECTS 8`

Up to this number, Murasaki can handle the interrupt objects. This is summation of the interrupt generation peripheral like UART1, UART2, SPI1, SPI2,... except EXTI.

Usually, 8 is enough.

#### 5.4.2.4 `#define NUM_OF_EXTI_OBJECTS 8`

Up to this number, Murasaki can handle the interrupt objects.

Usually, 8 is enough.

#### 5.4.2.5 `#define PLATFORM_CONFIG_DEBUG_BUFFER_SIZE 4096`

The circular buffer array length to copy the formatted strings before transmitting through the uart.

To override the definition here, define same macro inside [platform\\_config.hpp](#).

#### 5.4.2.6 `#define PLATFORM_CONFIG_DEBUG_LINE_SIZE 256`

The array length to store the formatted string. Note that this array is a private instance variable. Then, it will occupy the memory where the class is instantiated. For example, if an object is instantiated in the heap, this line buffer will be reserved in the heap.

If the class is instantiated on the stack, the buffer will be reserved in the stack.

To override the definition here, define same macro inside [platform\\_config.hpp](#).

#### 5.4.2.7 `#define PLATFORM_CONFIG_DEBUG_SERIAL_TIMEOUT (murasaki::kwmsIndefinitely)`

By default, there is no timeout. Wait for eternally.

To override the definition here, define same macro inside [platform\\_config.hpp](#).

#### 5.4.2.8 `#define PLATFORM_CONFIG_DEBUG_TASK_PRIORITY murasaki::ktpHigh`

The priority of the `murasaki::Debugger` internal task. To output the logging data as fast as possible, the debug task have to have relatively high priority. In other hand, to yield the CPU to the critical tasks, it's priority have to be smaller than the max priority.

To override the definition here, define same macro inside [platform\\_config.hpp](#).

#### 5.4.2.9 `#define PLATFORM_CONFIG_DEBUG_TASK_STACK_SIZE 256`

The `murasaki::Debugger` class has internal task to handle its FIFO buffer.

To override the definition here, define same macro inside [platform\\_config.hpp](#).



## 5.4.3 Enumeration Type Documentation

5.4.3.1 enum `murasaki::AdcStatus`

Enumerator

***kasOK*** No error.

5.4.3.2 enum `murasaki::CodecChannel`

Codec channels are codec dependent. Thus, channels are not hard coded as member function, but coded as parameter of the member function.

Enumerator

***kccLineInput*** kccLineInput

***kccMicInput*** kccMicInput Microphone Input

***kccAuxInput*** kccAuxInput Auxiliary Input

***kccLineOutput*** kccLineOutput

***kccHeadphoneOutput*** kccHpOutput Headphone Output

5.4.3.3 enum `murasaki::I2cStatus`

This enums represents the return status from the I2C class method.

In a single master controller system, you need to care only `ki2csNak` and `ki2csTimeOut`. Other errors may be caused by multiple master systems.

The `ki2csNak` is returned when one of two happens :

- The slave device terminated transfer.
- No slave device responded to the address specified by the master device.

The `ki2csTimeOut` is returned when the slave device stretched transfer too long.

The `ki2csArbitrationLost` is returned when another master won the arbitration. Usually, the master has to re-try the transfer after a certain waiting period.

The `ki2csBussError` is a fatal condition. In the master mode, it could be a problem with other devices. The root cause is not deterministic. Probably it is a hardware problem.

Enumerator

***ki2csOK*** ki2csOK

***ki2csTimeOut*** Master mode error. No response from device.

***ki2csNak*** Master mode error. Device answers NAK.

***ki2csBussError*** Master&Slave mode error. START/STOP condition at irregular location.

***ki2csArbitrationLost*** Master&Slave mode error. Lost arbitration against other master device.

***ki2csOverrun*** Slave mode error. Overrun or Underrun was detected.

***ki2csDMA*** Some error detected in DMA module.

***ki2csUnknown*** Unknown error.

#### 5.4.3.4 enum `murasaki::InterruptStatus`

##### Enumerator

***kisOK*** kisOK Released correctly.  
***kisTimeOut*** kisTimeOut Time out happen

#### 5.4.3.5 enum `murasaki::SpiClockPhase`

This enum represents the setting of the SPI PHA bit of the master configuration. The PHA setting 0 and 1 is LatchThenShift and ShiftThenLatch, respectively.

##### Enumerator

***ksphLatchThenShift*** kscpLatchThenShift PHA=0. The first edge is latching. The second edge is shifting.  
***ksphShiftThenLatch*** kscpShiftThenLatch PHA = 1. The first edge is shifting. The second edge is latching.

#### 5.4.3.6 enum `murasaki::SpiClockPolarity`

This enum represents the setting of the SPI POL bit of the master configuration. The POL setting 0/1 is RiseThenFall and Fall thenRise respectively.

##### Enumerator

***kspoRiseThenFall*** kscpRiseThenFall POL = 0  
***kspoFallThenRise*** kscpFallThenrise POL = 1

#### 5.4.3.7 enum `murasaki::SpiStatus`

This enums represents the return status of from the SPI class method.

kspisModeFault is returned when the NSS pins are asserted. Note that the Murasaki library doesn't support the Multi-master SPI operation. So, this is a fatal condition.

kpisOverflow and the kpisDMA are fatal conditions. These can be the problem of the lower driver problem.

##### Enumerator

***kspisOK*** ki2csOK  
***kspisTimeOut*** Master mode error. No response from device.  
***kspisModeFault*** SPI mode fault error. Two master corrsion.  
***kspisModeCRC*** CRC protocol error.  
***kspisOverflow*** Over run.  
***kspisFrameError*** Error on TI frame mode.  
***kspisDMA*** DMA error.  
***kspisErrorFlag*** Other error flag.  
***kspisAbort*** Problem in abort process. No way to recover.  
***kspisUnknown*** Unknown error.

#### 5.4.3.8 enum murasaki::SyslogFacility

These are independent facilities to filter the Syslog message output. Each module should specify the appropriate facility.

Internally, these values are used as a bit position in the mask.

##### Enumerator

- kfaNone** Disable all facility.
- kfaKernel** kfaKernel is specified when the message is bound with the kernel issue.
- kfaSerial** kfaSerial is specified when the message is from the serial module.
- kfaSpiMaster** kfaSpi is specified when the message is from the SPI master module
- kfaSpiSlave** kfaSpi is specified when the message is from the SPI slave module
- kfaI2cMaster** kfaI2c is specified when the message is from the I2C master module.
- kfaI2cSlave** kfaI2c is specified when the message is from the I2C slave module.
- kfaAudio** kfaI2c is specified when the message is from the Audio module.
- kfaI2s** kfaI2s is specified when the message is from the I2S module
- kfaSai** kfaSai is specified when the message is from the SAI module.
- kfaLog** kfaLog is specified when the message is from the logger and debugger module.
- kfaAudioCodec** kfaAudioCodec is specified when the message is from the Audio Codec module
- kfaEncoder** kfaEncoder is specified when the message is from the Encoder module.
- kfaAdc** kfaAdc is specified when the message is from the [Adc](#) module.
- kfaExti** kfaExti is specified when the message is from the [Exti](#) module.
- kfaUser0** User defined facility.
- kfaUser1** User defined facility.
- kfaUser2** User defined facility.
- kfaUser3** User defined facility.
- kfaUser4** User defined facility.
- kfaUser5** User defined facility.
- kfaUser6** User defined facility.
- kfaUser7** User defined facility.
- kfaAll** Enable all facility.

#### 5.4.3.9 enum murasaki::SyslogSeverity

The lower value is the more serious condition.

##### Enumerator

- kseEmergency** kseEmergency means the system is unusable.
- kseAlert** kseAlert means some acution must be taken immediately.
- kseCritical** kseCritical means critical condition.
- kseError** kseError means error conditions.
- kseWarning** kseWarning means warning condition.
- kseNotice** kseNotice means normal but significant condition.
- kseInfomational** kseInfomational means infomational message.
- kseDebug** kseDebug means debug-level message

#### 5.4.3.10 enum `murasaki::TaskPriority`

The task class priority have to be speicified by this enum class. This is essential to avoid the imcompatibility with cmsis-os which uses negative priority while FreeRTOS uses positive.

##### Enumerator

***ktpIdle*** ktpIdle

***ktpLow*** ktpLow

***ktpBelowNormal*** ktpBelowNormal is for the relatively low priority task.

***ktpNormal*** ktpNormal is for the default processing.

***ktpAboveNormal*** ktpAboveNormal is for the relatively high priority task.

***ktpHigh*** ktpHigh is considered for the debug task.

***ktpRealtime*** ktpRealtime is dedicated for the realtime signal processing.

#### 5.4.3.11 enum `murasaki::UartHardwareFlowControl`

This is dedicated to the [UartStrategy](#) class.

##### Enumerator

***kuhfcNone*** No hardware flow control.

***kuhfcCts*** Control CTS, but RTS.

***kuhfcRts*** Control RTS, but CTS.

***kuhfcCtsRts*** Control Both CTS and RTS.

#### 5.4.3.12 enum `murasaki::UartStatus`

The Parity error and the Frame error may occur when the user connects DCT/DTE by different communication setting.

The nose on the line may cause a Noise error.

The overrun may cause when the DMA is too slow, or handshake is not working well.

The DMA error may cause some problem inside HAL.

##### Enumerator

***kursOK*** No error.

***kursTimeOut*** Time out during transmission / receive.

***kursParity*** Parity error.

***kursNoise*** Error by Noise.

***kursFrame*** Frame error.

***kursOverrun*** Overrun error.

***kursDMA*** Error inside DMA module.

#### 5.4.3.13 enum murasaki::UartTimeout

The idle line time out is a dedicated function of the STM32 peripherals. The interrupt happens when the received data is discontinued a certain time.

##### Enumerator

***kutNoldleTimeout*** kutNoldleTimeout is specified when API should has normal timeout.

***kutIdleTimeout*** kutIdleTimeout is specified when API should time out by Idle line

#### 5.4.3.14 enum murasaki::WaitMilliSeconds : uint32\_t

An uint32\_t derived type for specifying wait duration. The integer value represents the waiting duration by milliseconds. Usually, a value of this type is passed to some functions as parameter. There are two special cases.

kwmsPolling means the function returns immediately regardless of the waited event. In other words, with this parameter, function causes a time out immediately. Some function may provide the way to know what was the status of the waited event. But some may not.

kwmsIndefinitely means the function does not cause a timeout.

##### Enumerator

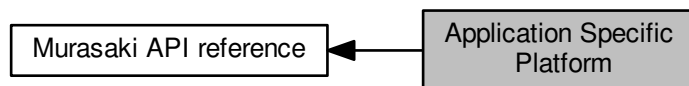
***kwmsPolling*** Not waiting. Immediate timeout.

***kwmsIndefinitely*** Wait forever.

## 5.5 Application Specific Platform

Variables to control the hardware.

Collaboration diagram for Application Specific Platform:



### Classes

- struct [murasaki::Platform](#)  
*Custom aggregation struct for user platform.*

### Functions

- void [InitPlatform](#) ()  
*Initialize the platform variables.*
- void [ExecPlatform](#) ()  
*The body of the real application.*
- void [CustomAssertFailed](#) (uint8\_t \*file, uint32\_t line)  
*Hook for the assert\_failure() in main.c.*
- void [CustomDefaultHandler](#) ()  
*Hook for the default exception handler. Never return.*
- void [HAL\\_UART\\_TxCpltCallback](#) (UART\_HandleTypeDef \*huart)  
*Essential to sync up with UART.*
- void [HAL\\_UART\\_RxCpltCallback](#) (UART\_HandleTypeDef \*huart)  
*Essential to sync up with UART.*
- void [HAL\\_UART\\_ErrorCallback](#) (UART\_HandleTypeDef \*huart)  
*Optional error handling of UART.*
- void [HAL\\_SPI\\_TxRxCpltCallback](#) (SPI\_HandleTypeDef \*hspl)  
*Essential to sync up with SPI.*
- void [HAL\\_SPI\\_ErrorCallback](#) (SPI\_HandleTypeDef \*hspl)  
*Optional error handling of SPI.*
- void [HAL\\_I2C\\_MasterTxCpltCallback](#) (I2C\_HandleTypeDef \*hi2c)  
*Essential to sync up with I2C.*
- void [HAL\\_I2C\\_SlaveTxCpltCallback](#) (I2C\_HandleTypeDef \*hi2c)  
*Essential to sync up with I2C.*
- void [HAL\\_I2C\\_ErrorCallback](#) (I2C\_HandleTypeDef \*hi2c)  
*Optional error handling of I2C.*
- void [HAL\\_SAI\\_RxHalfCpltCallback](#) (SAI\_HandleTypeDef \*hsai)  
*Optional SAI interrupt handler at buffer transfer halfway.*
- void [HAL\\_SAI\\_RxCpltCallback](#) (SAI\_HandleTypeDef \*hsai)

- Optional SAI interrupt handler at buffer transfer complete.*
- void `HAL_SAI_ErrorCallback` (SAI\_HandleTypeDef \*hsai)  
*Optional SAI error interrupt handler.*
- void `HAL_I2S_RxCpltCallback` (I2S\_HandleTypeDef \*hi2s)  
*Optional I2S interrupt handler at buffer transfer halfway.*
- void `HAL_I2S_RxCpltCallback` (I2S\_HandleTypeDef \*hi2s)  
*Optional I2S interrupt handler at buffer transfer complete.*
- void `HAL_I2S_ErrorCallback` (I2S\_HandleTypeDef \*hi2s)  
*Optional I2S error interrupt handler.*
- void `HAL_GPIO_EXTI_Callback` (uint16\_t GPIO\_Pin)  
*Optional interrupt handling of EXTI.*

## Variables

- Debugger \* `murasaki::debugger`  
*A global variable to provide the debugging function.*

### 5.5.1 Detailed Description

The typical usage of these variables can be seen below. First of all, a .cpp file has to include `murasaki.hpp`.

```
#include "murasaki.hpp"
```

And then, define the `murasaki::debugger` in the global context. Note that this is essential to use certain debug macros.

The definition of the `murasaki::platform` is optional. But it is recommended declaring for the ease of reading.

```
murasaki::Debugger * murasaki::debugger;
murasaki::Platform * murasaki::platform;
```

Finally, initialize the `murasaki::debugger` and `murasaki::platform`. Again, the `murasaki::debugger` is essential to use the debug macro. The debug macros are used inside the murasaki class library. Then, it is mandatory to initialize the debugger member variable.

The following code fragment initializes only the debugger related member variables. Also, the `murasaki::Platform` variable is refereed.

The platform.uart\_console member variable hooks a murasaki::AbstractUart class variable. In this sample, The `murasaki::Uart` class is instantiated. The Uart constructor receives the pointer to the UART\_HandleTypeDef. Usually, the UART\_HandleTypeDef variable is generated by CubeIDE. For example, "huart3" variable in the `main.c` file.

The platform.logger member variable hooks a murasaki::AbstractLogger variable. In this example, `murasaki::UartLogger` class variable is instantiated.

Finally, the debugger variable is initialized. The `murasaki::Debugger` constructor receives murasaki::AbstractLogger \* type.

```
void InitPlatform(UART_HandleTypeDef * uart_handle)
{
    murasaki::platform.uart_console = new murasaki::Uart(uart_handle);
    murasaki::platform.logger = new murasaki::UartLogger(murasaki::platform.
        uart_console);

    murasak::debugger = new murasaki::Debugger(murasaki::platform.logger
        );
}
```

### 5.5.2 Function Documentation

#### 5.5.2.1 void CustomAssertFailed ( uint8\_t \* file, uint32\_t line )

**Parameters**

<i>file</i>	Name of the source file where assertion happen
<i>line</i>	Number of the line where assertion happen

This routine provides a custom hook for the assertion inside STM32Cube HAL. All assertion raised in HAL will be redirected here.

```
1 void assert_failed(uint8_t* file, uint32_t line)
2 {
3     CustomAssertFailed(file, line);
4 }
```

By default, this routine output a message with location information to the debugger console.

**5.5.2.2 void CustomDefaultHandler ( )**

An entry of the exception. Especially for the Hard Fault exception. In this function, the Stack pointer just before exception is retrieved and pass as the first parameter of the [PrintFaultResult\(\)](#).

Note : To print the correct information, this function have to be Jumped in from the exception entry without any data push to the stack. To avoid the pushing extra data to stack or making stack frame, Compile the program without debug information and with certain optimization level, when you investigate the Hard Fault.

For example, the start up code for the Nucleo-L152RE is startup\_stm32l152xe.s. This file is generated by CubeIDE. This file has default handler as like this:

```
1 .section .text.Default_Handler,"ax",%progbits
2     Default_Handler:
3 Infinite_Loop:
4     b Infinite_Loop
```

This code can be modified to call CustomDefaultHandler as like this :

```
1 .global CustomDefaultHandler
2 .section .text.Default_Handler,"ax",%progbits
3 Default_Handler:
4     bl CustomDefaultHandler
5 Infinite_Loop:
6     b Infinite_Loop
```

While it is declared as function prototype, the CustomDefaultHandler is just a label. Do not call from user application.

**5.5.2.3 void ExecPlatform ( )**

The body function of the murasaki application. Usually this function is called from the [StartDefaultTask\(\)](#) of the [main.c](#).

This function is invoked only once, and never return. See [InitPlatform\(\)](#) as calling sample.

By default, it toggles LED as sample program. This function can be customized freely.

**5.5.2.4 void HAL\_GPIO\_EXTI\_Callback ( uint16\_t GPIO\_Pin )**



## Parameters

<i>GPIO_Pin</i>	Pin number from 0 to 31
-----------------	-------------------------

This is called from inside of HAL when an EXTI is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default error interrupt call back.

The GPIO\_Pin is the number of Pin. For example, if a programmer set the pin name by CubeIDE as FOO, the macro to identify that EXTI is FOO\_Pin

5.5.2.5 void HAL\_I2C\_ErrorCallback ( I2C\_HandleTypeDef \* *hi2c* )

## Parameters

<i>hi2c</i>	
-------------	--

This is called from inside of HAL when an I2C error interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default error interrupt call back.

In this call back, the uart device handle have to be passed to the `murasaki::I2c::HandleError()` function.

5.5.2.6 void HAL\_I2C\_MasterTxCpltCallback ( I2C\_HandleTypeDef \* *hi2c* )

## Parameters

<i>hi2c</i>	
-------------	--

This is called from inside of HAL when an I2C transmission done interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default TX interrupt call back.

In this call back, the uart device handle have to be passed to the `murasaki::I2c::TransmitCompleteCallback()` function.

5.5.2.7 void HAL\_I2C\_SlaveTxCpltCallback ( I2C\_HandleTypeDef \* *hi2c* )

## Parameters

<i>hi2c</i>	
-------------	--

This is called from inside of HAL when an I2C transmission done interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt

happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default TX interrupt call back.

In this call back, the I2C slave device handle have to be passed to the [murasaki::I2cSlave::TransmitComplete↔Callback\(\)](#) function.

#### 5.5.2.8 void HAL\_I2S\_ErrorCallback ( I2S\_HandleTypeDef \* *hi2s* )

##### Parameters

<i>hi2s</i>	Handler of the I2S device.
-------------	----------------------------

The error have to be forwarded to [murasaki::DuplexAudio::HandleError\(\)](#). Note that DuplexAudio::HandleError() trigger a hard fault. So, never return.

#### 5.5.2.9 void HAL\_I2S\_RxCpltCallback ( I2S\_HandleTypeDef \* *hi2s* )

##### Parameters

<i>hi2s</i>	Handler of the I2S device.
-------------	----------------------------

Invoked after I2S RX DMA complete interrupt is at halfway. This interrupt have to be forwarded to the [murasaki↔::DuplexAudio::ReceiveCallback\(\)](#). The second parameter of the ReceiveCallback() have to be 1 which mean the complete interrupt.

#### 5.5.2.10 void HAL\_I2S\_RxHalfCpltCallback ( I2S\_HandleTypeDef \* *hi2s* )

##### Parameters

<i>hi2s</i>	Handler of the I2S device.
-------------	----------------------------

Invoked after I2S RX DMA complete interrupt is at halfway. This interrupt have to be forwarded to the [murasaki↔::DuplexAudio::ReceiveCallback\(\)](#). The second parameter of the ReceiveCallback() have to be 0 which mean the halfway interrupt.

#### 5.5.2.11 void HAL\_SAI\_ErrorCallback ( SAI\_HandleTypeDef \* *hsai* )

##### Parameters

<i>hsai</i>	Handler of the SAI device.
-------------	----------------------------

The error have to be forwarded to [murasaki::DuplexAudio::HandleError\(\)](#). Note that DuplexAudio::HandleError() trigger a hard fault. So, never return.

#### 5.5.2.12 void HAL\_SAI\_RxCpltCallback ( SAI\_HandleTypeDef \* *hsai* )

##### Parameters

<i>hsai</i>	Handler of the SAI device.
-------------	----------------------------

Invoked after SAI RX DMA complete interrupt is at halfway. This interrupt have to be forwarded to the `murasaki::DuplexAudio::ReceiveCallback()`. The second parameter of the `ReceiveCallback()` have to be 1 which mean the complete interrupt.

#### 5.5.2.13 void HAL\_SAI\_RxHalfCpltCallback ( SAI\_HandleTypeDef \* *hsai* )

##### Parameters

<i>hsai</i>	Handler of the SAI device.
-------------	----------------------------

Invoked after SAI RX DMA complete interrupt is at halfway. This interrupt have to be forwarded to the `murasaki::DuplexAudio::ReceiveCallback()`. The second parameter of the `ReceiveCallback()` have to be 0 which mean the halfway interrupt.

#### 5.5.2.14 void HAL\_SPI\_ErrorCallback ( SPI\_HandleTypeDef \* *hspi* )

##### Parameters

<i>hspi</i>	
-------------	--

This is called from inside of HAL when an SPI error interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default error interrupt call back.

In this call back, the uart device handle have to be passed to the `murasaki::Uart::HandleError()` function.

#### 5.5.2.15 void HAL\_SPI\_TxRxCpltCallback ( SPI\_HandleTypeDef \* *hspi* )

##### Parameters

<i>hspi</i>	
-------------	--

This is called from inside of HAL when an SPI transfer done interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default TX/RX interrupt call back.

In this call back, the SPI device handle have to be passed to the `murasaki::Spi::TransmitAndReceiveCompleteCallback()` function.

#### 5.5.2.16 void HAL\_UART\_ErrorCallback ( UART\_HandleTypeDef \* *huart* )

##### Parameters

<i>huart</i>	
--------------	--

This is called from inside of HAL when an UART error interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default error interrupt call back.

In this call back, the uart device handle have to be passed to the [murasaki::Uart::HandleError\(\)](#) function.

#### 5.5.2.17 void HAL\_UART\_RxCpltCallback ( UART\_HandleTypeDef \* huart )

##### Parameters

<i>huart</i>	
--------------	--

This is called from inside of HAL when an UART receive done interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default RX interrupt call back.

In this call back, the uart device handle have to be passed to the [murasaki::Uart::ReceiveCompleteCallback\(\)](#) function.

#### 5.5.2.18 void HAL\_UART\_TxCpltCallback ( UART\_HandleTypeDef \* huart )

##### Parameters

<i>huart</i>	
--------------	--

This is called from inside of HAL when an UART transmission done interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default TX interrupt call back.

In this call back, the uart device handle have to be passed to the [murasaki::Uart::TransmissionCompleteCallback\(\)](#) function.

#### 5.5.2.19 void InitPlatform ( )

The [murasaki::platform](#) variable is an interface between the application program and HAL / RTOS. To use it correctly, the initialization is needed before any activity of murasaki client.

```
1 void StartDefaultTask(void const * argument)
2 {
3     InitPlatform();
4     ExecPlatform();
5 }
```

This function have to be invoked from the [StartDefaultTask\(\)](#) of the [main.c](#) only once to initialize the platform variable.

### 5.5.3 Variable Documentation

#### 5.5.3.1 murasaki::Debugger \* murasaki::debugger

This variable is declared by murasaki platform, but not instantiated. To make it happen, a programmer has to make a variable and initialize it explicitly. Otherwise, some debug utility/macro may cause link error because [urasaki::debugger](#) is called from these utility/macros.

## 5.6 Abstract Classes

Generic classes as template of the concrete class.

Collaboration diagram for Abstract Classes:



### Classes

- class [murasaki::AdcStrategy](#)  
*Synchronized, blocking ADC converter Strategy.*
- class [murasaki::AudioCodecStrategy](#)  
*abstract audio CODEC controller.*
- class [murasaki::AudioPortAdapterStrategy](#)  
*Strategy of the audio device adaptor..*
- class [murasaki::BitInStrategy](#)  
*Definition of the root class of bit input.*
- class [murasaki::BitOutStrategy](#)  
*Definition of the root class of bit output.*
- class [murasaki::FifoStrategy](#)  
*Basic FIFO without thread safe.*
- class [murasaki::I2cCallbackStrategy](#)  
*Definition of the root class of I2C .*
- class [murasaki::I2cMasterStrategy](#)  
*Definition of the root class of I2C master.*
- class [murasaki::I2cSlaveStrategy](#)  
*Definition of the root class of I2C Slave.*
- class [murasaki::InterruptStrategy](#)  
*Abstract interrupt class.*
- class [murasaki::LoggerStrategy](#)  
*Abstract class for logging.*
- class [murasaki::PeripheralStrategy](#)  
*Mother of all peripheral class.*
- class [murasaki::QuadratureEncoderStrategy](#)  
*Strategy class for the quadrature encoder.*
- class [murasaki::SpiCallbackStrategy](#)  
*Root class of the SPI.*
- class [murasaki::SpiMasterStrategy](#)  
*Root class of the SPI master.*
- class [murasaki::SpiSlaveAdapterStrategy](#)  
*Definition of the root class of SPI slave adapter.*
- class [murasaki::SpiSlaveStrategy](#)  
*Root class of the SPI slave.*
- class [murasaki::TaskStrategy](#)  
*A mother of all tasks.*
- class [murasaki::UartStrategy](#)  
*Definition of the root class of UART.*

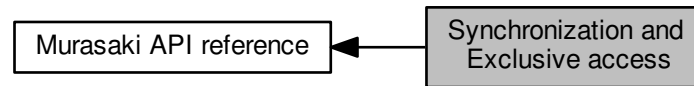
### 5.6.1 Detailed Description

Usually, application doesn't instantiate these classes. But pointer may be declared as abstract class as generic placeholder.

## 5.7 Synchronization and Exclusive access

Sync between the task and interrupt. Make the resources thread safe.

Collaboration diagram for Synchronization and Exclusive access:



### Classes

- class [murasaki::CriticalSection](#)  
*A critical section for task context.*
- class [murasaki::Synchronizer](#)  
*Synchronization class between a task and interrupt. This class provide the synchronization between a task and interrupt.*

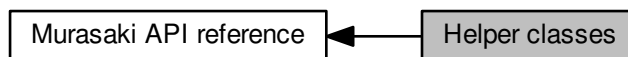
### 5.7.1 Detailed Description

These classes are used as parts of the other classes.

## 5.8 Helper classes

Classes to support the `murasaki-class`.

Collaboration diagram for Helper classes:



### Classes

- struct `murasaki::GPIO_type`  
*A structure to en-group the GPIO port and GPIO pin.*
- class `murasaki::CallbackRepositorySingleton`  
*Repository of the interrupt handling objects.*
- class `murasaki::DebuggerFifo`  
*FIFO with thread safe.*
- struct `murasaki::LoggingHelpers`  
*A stracture to engroup the logging tools.*
- class `murasaki::DebuggerUart`  
*Logging dedicated UART class.*
- class `murasaki::ExtiCallbackRepositorySingleton`  
*Repository of the EXTI handling objects.*

### Functions

- void \* `operator new` (std::size\_t size)  
*Allocate a memory piece with given size.*
- void \* `operator new[]` (std::size\_t size)  
*Allocate a memory piece with given size.*
- void `operator delete` (void \*ptr)  
*Deallocate the given memory.*
- void `operator delete[]` (void \*ptr)  
*Deallocate the given memory.*

#### 5.8.1 Detailed Description

These classes are not used by customer.

#### 5.8.2 Function Documentation

##### 5.8.2.1 void operator delete ( void \* ptr )



## Parameters

<i>ptr</i>	Pointer to the memory to deallocate
------------	-------------------------------------

## Returns

Allocated memory in FreeRTOS heap. Null mean fail to allocate.

5.8.2.2 void operator delete[] ( void \* *ptr* )

## Parameters

<i>ptr</i>	Pointer to the memory to deallocate
------------	-------------------------------------

## Returns

Allocated memory in FreeRTOS heap. Null mean fail to allocate.

5.8.2.3 void\* operator new ( std::size\_t *size* )

## Parameters

<i>size</i>	Size of the memory to allocate [byte]
-------------	---------------------------------------

## Returns

Allocated memory in FreeRTOS heap. Null mean fail to allocate.

5.8.2.4 void\* operator new[] ( std::size\_t *size* )

## Parameters

<i>size</i>	Size of the memory to allocate [byte]
-------------	---------------------------------------

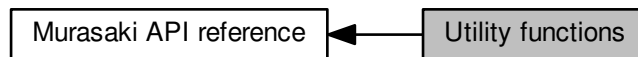
## Returns

Allocated memory in FreeRTOS heap. Null mean fail to allocate.

## 5.9 Utility functions

Collection of the useful functions.

Collaboration diagram for Utility functions:



### Functions

- void `murasaki::I2cSearch` (`murasaki::I2cMasterStrategy *master`)  
*I2C device search function.*
- void `murasaki::InitCycleCounter` ()  
*Initialize and start the cycle counter.*
- unsigned int `murasaki::GetCycleCounter` ()  
*Obtain the current cycle count of CYCCNT register.*

#### 5.9.1 Detailed Description

#### 5.9.2 Function Documentation

##### 5.9.2.1 unsigned int `murasaki::GetCycleCounter` ( )

#### Returns

current core cycle.

Regarding CORTEX-M0 and M0+, there is no CYCCNT. Thus, we do nothing in this function.

Programmer can override default function because this function is weakly bound.

##### 5.9.2.2 void `murasaki::I2cSearch` ( `murasaki::I2cMasterStrategy * master` )

#### Parameters

<code>master</code>	Pointer to the I2C master controller object.
---------------------	--

Poll all device address and check the response. If no response(NAK), there is no device.

### 5.9.2.3 void murasaki::InitCycleCounter ( )

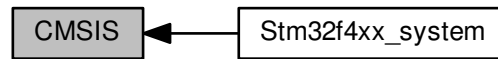
This cycle counter ( CYCNT ) is implemented inside CORTEX-Mx core. To implement or not is up to the SoC vender.

Regarding CORTEX-M0 and M0+, there is no CYCCNT. THus, we do noting in this function.

Programmer can override default function because this funciton is weakly bound.

## 5.10 CMSIS

Collaboration diagram for CMSIS:



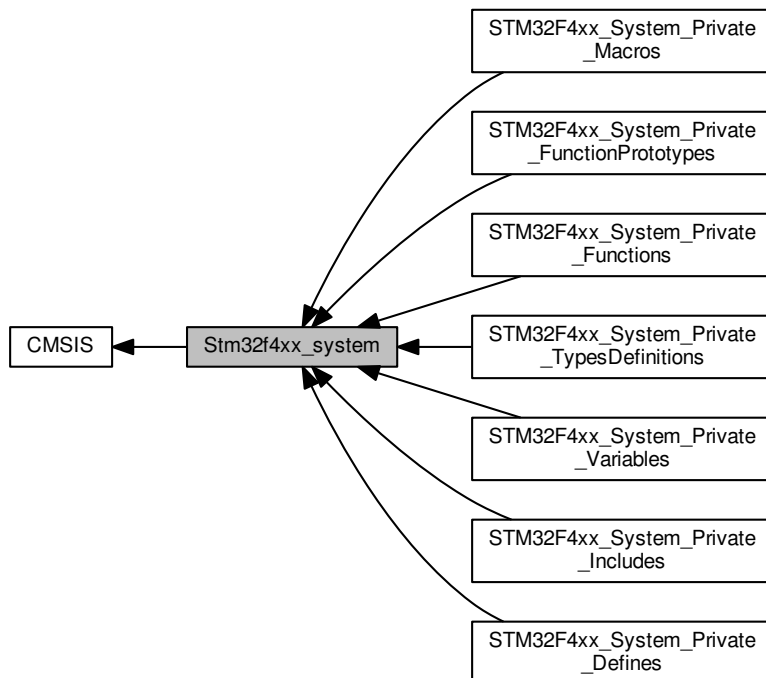
### Modules

- [Stm32f4xx\\_system](#)

### 5.10.1 Detailed Description

## 5.11 Stm32f4xx\_system

Collaboration diagram for Stm32f4xx\_system:



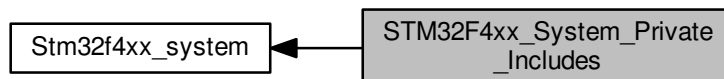
### Modules

- [STM32F4xx\\_System\\_Private\\_Includes](#)
- [STM32F4xx\\_System\\_Private\\_TypesDefinitions](#)
- [STM32F4xx\\_System\\_Private\\_Defines](#)
- [STM32F4xx\\_System\\_Private\\_Macros](#)
- [STM32F4xx\\_System\\_Private\\_Variables](#)
- [STM32F4xx\\_System\\_Private\\_FunctionPrototypes](#)
- [STM32F4xx\\_System\\_Private\\_Functions](#)

### 5.11.1 Detailed Description

## 5.12 STM32F4xx\_System\_Private\_Includes

Collaboration diagram for STM32F4xx\_System\_Private\_Includes:



### Macros

- `#define HSE_VALUE ((uint32_t)25000000)`
- `#define HSI_VALUE ((uint32_t)16000000)`

#### 5.12.1 Detailed Description

#### 5.12.2 Macro Definition Documentation

##### 5.12.2.1 `#define HSE_VALUE ((uint32_t)25000000)`

Default value of the External oscillator in Hz

##### 5.12.2.2 `#define HSI_VALUE ((uint32_t)16000000)`

Value of the Internal oscillator in Hz

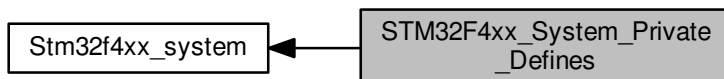
## 5.13 STM32F4xx\_System\_Private\_TypesDefinitions

Collaboration diagram for STM32F4xx\_System\_Private\_TypesDefinitions:



## 5.14 STM32F4xx\_System\_Private\_Defines

Collaboration diagram for STM32F4xx\_System\_Private\_Defines:



### Macros

- `#define VECT_TAB_OFFSET 0x00`

#### 5.14.1 Detailed Description

#### 5.14.2 Macro Definition Documentation

##### 5.14.2.1 `#define VECT_TAB_OFFSET 0x00`

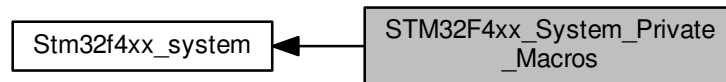
< Uncomment the following line if you need to use external SRAM or SDRAM as data memory

< Uncomment the following line if you need to relocate your vector Table in Internal SRAM. Vector Table base offset field. This value must be a multiple of 0x200.



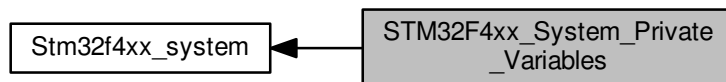
## 5.15 STM32F4xx\_System\_Private\_Macros

Collaboration diagram for STM32F4xx\_System\_Private\_Macros:



## 5.16 STM32F4xx\_System\_Private\_Variables

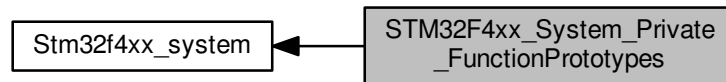
Collaboration diagram for STM32F4xx\_System\_Private\_Variables:



### 5.16.1 Detailed Description

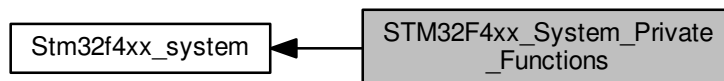
## 5.17 STM32F4xx\_System\_Private\_FunctionPrototypes

Collaboration diagram for STM32F4xx\_System\_Private\_FunctionPrototypes:



## 5.18 STM32F4xx\_System\_Private\_Functions

Collaboration diagram for STM32F4xx\_System\_Private\_Functions:



### Functions

- void [SystemInit](#) (void)  
*Setup the microcontroller system Initialize the FPU setting, vector table location and External memory configuration.*
- void [SystemCoreClockUpdate](#) (void)  
*Update SystemCoreClock variable according to Clock Register Values. The SystemCoreClock variable contains the core clock (HCLK), it can be used by the user application to setup the SysTick timer or configure other parameters.*

### 5.18.1 Detailed Description

### 5.18.2 Function Documentation

#### 5.18.2.1 void SystemCoreClockUpdate ( void )

#### Note

Each time the core clock (HCLK) changes, this function must be called to update SystemCoreClock variable value. Otherwise, any configuration based on this variable will be incorrect.

- The system frequency computed by this function is not the real frequency in the chip. It is calculated based on the predefined constant and the selected clock source:

- If SYSCLK source is HSI, SystemCoreClock will contain the [HSI\\_VALUE\(\\*\)](#)
- If SYSCLK source is HSE, SystemCoreClock will contain the [HSE\\_VALUE\(\\*\\*\)](#)
- If SYSCLK source is PLL, SystemCoreClock will contain the [HSE\\_VALUE\(\\*\\*\)](#) or [HSI\\_VALUE\(\\*\)](#) multiplied/divided by the PLL factors.

(\*) HSI\_VALUE is a constant defined in stm32f4xx\_hal\_conf.h file (default value 16 MHz) but the real value may vary depending on the variations in voltage and temperature.

(\*\*) HSE\_VALUE is a constant defined in stm32f4xx\_hal\_conf.h file (its value depends on the application requirements), user has to ensure that HSE\_VALUE is same as the real frequency of the crystal used. Otherwise, this function may have wrong result.

- The result of this function could be not correct when using fractional value for HSE crystal.

## Parameters

<i>None</i>	
-------------	--

## Return values

<i>None</i>	
-------------	--

## 5.18.2.2 void SystemInit ( void )

## Parameters

<i>None</i>	
-------------	--

## Return values

<i>None</i>	
-------------	--

## 6 Namespace Documentation

### 6.1 murasaki Namespace Reference

Personal [Platform](#) parts collection.

#### Classes

- class [Adau1361](#)  
*Audio Codec LSI class.*
- class [Adc](#)  
*STM32 dedicated ADC class.*
- class [AdcStrategy](#)  
*Synchronized, blocking ADC converter Strategy.*
- class [AudioCodecStrategy](#)  
*abstract audio CODEC controller.*
- class [AudioPortAdapterStrategy](#)  
*Strategy of the audio device adaptor..*
- class [BitIn](#)  
*General purpose bit input.*
- class [BitInStrategy](#)  
*Definition of the root class of bit input.*
- class [BitOut](#)  
*General purpose bit output.*
- class [BitOutStrategy](#)  
*Definition of the root class of bit output.*
- class [CallbackRepositorySingleton](#)  
*Repository of the interrupt handling objects.*
- class [CriticalSection](#)  
*A critical section for task context.*
- class [Debugger](#)  
*Debug class. Provides printf() style output for both task and ISR context.*
- class [DebuggerFifo](#)  
*FIFO with thread safe.*
- class [DebuggerUart](#)  
*Logging dedicated UART class.*
- class [DuplexAudio](#)  
*Stereo Audio is served by this class.*
- class [Exti](#)  
*EXTI wrapper class.*
- class [ExtiCallbackRepositorySingleton](#)  
*Repository of the EXTI handling objects.*
- class [FifoStrategy](#)  
*Basic FIFO without thread safe.*
- struct [GPIO\\_type](#)  
*A structure to en-group the GPIO port and GPIO pin.*
- class [I2cCallbackStrategy](#)  
*Definition of the root class of I2C .*
- class [I2cMaster](#)

- Thread safe, synchronous, and blocking IO. Encapsulating I2C master. Based on STM32Cube HAL driver and FreeRTOS.*

  - class [I2cMasterStrategy](#)

*Definition of the root class of I2C master.*
  - class [I2cSlave](#)

*Thread safe, synchronous and blocking IO. Encapsulating I2C slave. Based on STM32Cube HAL driver and FreeRTOS.*
  - class [I2cSlaveStrategy](#)

*Definition of the root class of I2C Slave.*
  - class [I2sPortAdapter](#)

*Adapter as I2S audio port.*
  - class [InterruptStrategy](#)

*Abstract interrupt class.*
  - class [LoggerStrategy](#)

*Abstract class for logging.*
  - struct [LoggingHelpers](#)

*A structure to engroup the logging tools.*
  - class [PeripheralStrategy](#)

*Mother of all peripheral class.*
  - struct [Platform](#)

*Custom aggregation struct for user platform.*
  - class [QuadratureEncoder](#)

*Quadrature Encoder class.*
  - class [QuadratureEncoderStrategy](#)

*Strategy class for the quadrature encoder.*
  - class [SaiPortAdapter](#)

*Adapter as SAI audio port.*
  - class [SimpleTask](#)

*An easy to use task class.*
  - class [SpiCallbackStrategy](#)

*Root class of the SPI.*
  - class [SpiMaster](#)

*Thread safe, synchronous and blocking IO. Encapsulating SPI master. Based on STM32Cube HAL driver and FreeRTOS.*
  - class [SpiMasterStrategy](#)

*Root class of the SPI master.*
  - class [SpiSlave](#)

*Thread safe, synchronous and blocking IO. Encapsulating SPI slave. Based on STM32Cube HAL driver and FreeRTOS.*
  - class [SpiSlaveAdapter](#)

*A spacificier of SPI slave.*
  - class [SpiSlaveAdapterStrategy](#)

*Definition of the root class of SPI slave adapter.*
  - class [SpiSlaveStrategy](#)

*Root class of the SPI slave.*
  - class [Synchronizer](#)

*Synchronization class between a task and interrupt. This class provide the synchronization between a task and interrupt.*
  - class [TaskStrategy](#)

*A mother of all tasks.*
  - class [Uart](#)

*Thread safe, synchronous and blocking IO. Concrete implementation of UART controller. Based on the STM32Cube HAL DMA Transfer.*

- class [UartLogger](#)

*Logging through an UART port.*

- class [UartStrategy](#)

*Definition of the root class of UART.*

## Enumerations

## Functions

- void [SetSyslogSererityThreshold](#) ([murasaki::SyslogSeverity](#) severity)  
*Set the syslog severity threshold.*
- void [SetSyslogFacilityMask](#) (uint32\_t mask)  
*Set the syslog facility mask.*
- void [AddSyslogFacilityToMask](#) ([murasaki::SyslogFacility](#) facility)  
*Add Syslog facility to the filter mask.*
- void [RemoveSyslogFacilityFromMask](#) ([murasaki::SyslogFacility](#) facility)  
*Remove Syslog facility to the filter mask.*
- bool [AllowedSyslogOut](#) ([murasaki::SyslogFacility](#) facility, [murasaki::SyslogSeverity](#) severity)  
*Check if given facility and severity message is allowed to output.*
- void [I2cSearch](#) ([murasaki::I2cMasterStrategy](#) \*master)  
*I2C device serach function.*
- void [InitCycleCounter](#) ()  
*Initialize and start the cycle counter.*
- unsigned int [GetCycleCounter](#) ()  
*Obtain the current cycle count of CYCCNT register.*

## Variables

- [Debugger](#) \* [debugger](#)  
*A global variable to provide the debugging function.*
- [Platform](#) [platform](#)  
*Grobal variable to provide the access to the platform component.*

### 6.1.1 Detailed Description

This namespace encloses personal collections of the software parts to create a "platform" of the software development. This specific collection is based on the STM32Cube HAL and FreeRTOS, both are generated by CubeIDE.

### 6.1.2 Function Documentation

#### 6.1.2.1 void [murasaki::AddSyslogFacilityToMask](#) ( [murasaki::SyslogFacility](#) *facility* )



## Parameters

<i>facility</i>	Allow this facility to output
-----------------	-------------------------------

See [AllowedSyslogOut](#) to understand when the message is out.

6.1.2.2 `bool murasaki::AllowedSyslogOut ( murasaki::SyslogFacility facility, murasaki::SyslogSeverity severity )`

## Parameters

<i>facility</i>	Message facility
<i>severity</i>	Message severity

## Returns

True if the message is allowed to out. False if not allowed.

By comparing internal severity threshold and facility mask, decide whether the message can be out or not.

If the severity is higher than or equal to `kseError`, the message is allowed to out.

If the severity is lower than `kseError`, the message is allowed to out only when :

- The severity is higher than or equal to the internal threshold
- The facility is "1" in the corresponding bit of the internal facility mask.

6.1.2.3 `void murasaki::RemoveSyslogFacilityFromMask ( murasaki::SyslogFacility facility )`

## Parameters

<i>facility</i>	Deny this facility to output
-----------------	------------------------------

See [AllowedSyslogOut](#) to understand when the message is out.

6.1.2.4 `void murasaki::SetSyslogFacilityMask ( uint32_t mask )`

## Parameters

<i>mask</i>	Facility bitmask. "1" allows the output of the corresponding facility
-------------	---

The parameter is not the facility. A bitmask. By default, the bitmask is `0xFFFFFFFF` which allows all facilities.

See [AllowedSyslogOut](#) to understand when the message is out.

6.1.2.5 `void murasaki::SetSyslogSererityThreshold ( murasaki::SyslogSeverity severity )`

#### Parameters

<i>severity</i>	
-----------------	--

Set the severity threshold. The message below this levels are ignored.

### 6.1.3 Variable Documentation

#### 6.1.3.1 `murasaki::Platform` `murasaki::platform`

This variable is declared by murasaki platform. But not instantiated. To make it happen, programmer have to make an variable and initilize it explicitly.

Note that the instantiation of this variable is optional. This is provided just of ease of read.

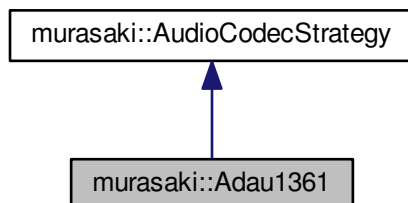
## 7 Class Documentation

### 7.1 `murasaki::Adau1361` Class Reference

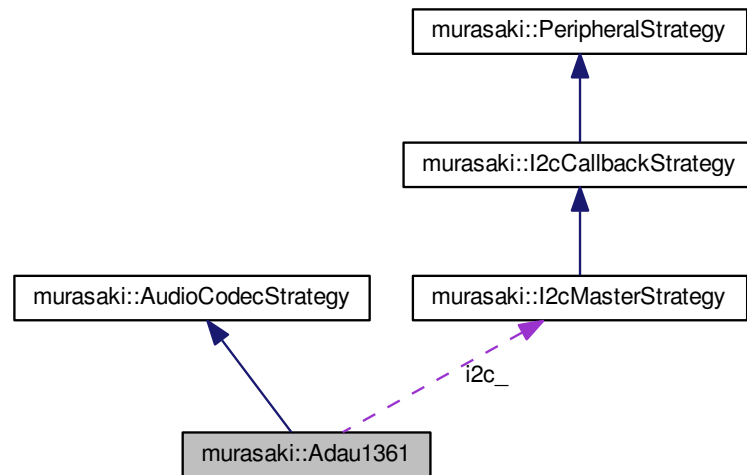
Audio Codec LSI class.

```
#include <adau1361.hpp>
```

Inheritance diagram for `murasaki::Adau1361`:



Collaboration diagram for murasaki::Adu1361:



#### Public Member Functions

- `Adu1361` (unsigned int fs, unsigned int master\_clock, `murasaki::I2cMasterStrategy` \*controller, unsigned int i2c\_device\_addr)  
*constructor.*
- virtual void `Start` (void)  
*Set up the ADAU1361 codec, and then, start the codec.*
- virtual void `SetGain` (`murasaki::CodecChannel` channel, float left\_gain, float right\_gain)  
*Set channel gain.*
- virtual void `Mute` (`murasaki::CodecChannel` channel, bool mute=true)  
*Mute the specific channel.*
- virtual void `SendCommand` (const uint8\_t command[], int size)  
*send one command to ADAU1361.*

#### Protected Member Functions

- virtual void `WaitPllLock` (void)  
*wait until PLL locks.*
- virtual void `ConfigurePll` (void)  
*Initialize the PLL with given fs and master clock.*
- virtual void `SetLineInputGain` (float left\_gain, float right\_gain, bool mute=false)  
*Set the line input gain and enable the relevant mixer.*
- virtual void `SetAuxInputGain` (float left\_gain, float right\_gain, bool mute=false)  
*Set the aux input gain and enable the relevant mixer.*
- virtual void `SetLineOutputGain` (float left\_gain, float right\_gain, bool mute=false)  
*Set the line output gain and enable the relevant mixer.*
- virtual void `SetHpOutputGain` (float left\_gain, float right\_gain, bool mute=false)  
*Set the headphone output gain and enable the relevant mixer.*
- virtual void `SendCommandTable` (const uint8\_t table[][3], int rows)  
*send one command to ADAU1361.*

## Additional Inherited Members

### 7.1.1 Detailed Description

Initialize the ADAU1361 codec based on the given parameter.

### 7.1.2 Constructor & Destructor Documentation

**7.1.2.1** `murasaki::Adau1361::Adau1361 ( unsigned int fs, unsigned int master_clock, murasaki::I2cMasterStrategy * controller, unsigned int i2c_device_addr )`

#### Parameters

<i>fs</i>	Sampling frequency[Hz]
<i>master_clock</i>	Input master clock frequency to the MCLK pin[Hz]
<i>controller</i>	Pass the I2C controller object.
<i>i2c_device_addr</i>	I2C device address. value range is from 0 to 127

initialize the internal variables. This constructor assumes the codec receive a master clock from outside. And output the I2C clocks as clock master.

The *fs* parameter is the sampling frequency of the CODEC in Hz. This parameter is limited as one of the following :

- 24000
- 32000
- 48000
- 96000
- 22050
- 44100
- 88200

The *master\_clock* parameter is the MCLK input to the ADAU1361 in Hz. This parameter must be one of followings :

- 8000000
- 12000000
- 13000000
- 14400000
- 19200000
- 19680000
- 19800000
- 24000000
- 26000000

- 27000000
- 12288000
- 24576000

Note : Only 8, 12, 13, 14.4, 12.288MHz are tested.

The analog signals are routed as following :

- Line In : LINN/RINN single ended.
- Aux In : LAUX/RAUX input
- LINE out : LOUTP/ROUTP single ended
- HP out : LHP/RHP

### 7.1.3 Member Function Documentation

**7.1.3.1** `virtual void murasaki::Adau1361::ConfigurePll ( void )` `[protected]`, `[virtual]`

At first, initialize the PLL based on the given fst and master clock. Then, setup the Converter sampling rate.

**7.1.3.2** `virtual void murasaki::Adau1361::Mute ( murasaki::CodecChannel channel, bool mute = true )` `[virtual]`

#### Parameters

<i>channel</i>	Channel to mute on / off
<i>mute</i>	On if true, off if false.

Implements [murasaki::AudioCodecStrategy](#).

**7.1.3.3** `virtual void murasaki::Adau1361::SendCommand ( const uint8_t command[], int size )` `[virtual]`

Service function for the ADAu1361 board implementer.

#### Parameters

<i>command</i>	command data array. It have to have register address of ADAU1361 in first two bytes.
<i>size</i>	number of bytes in the command, including the regsiter address.

Send one complete command to ADAU3161 by I2C. In the typical case, the command length is 3.

- `command[0]` : USB of the register address. 0x40.
- `command[1]` : LSB of the register address.
- `command[2]` : Value to right the register.

Implements [murasaki::AudioCodecStrategy](#).

**7.1.3.4** `virtual void murasaki::Adau1361::SendCommandTable ( const uint8_t table[ ][3], int rows )` [protected], [virtual]

#### Parameters

<i>table</i>	command table. All commands are stored in one row. Each row has only 1 byte data after reg address.
<i>rows</i>	number of the rows in the table.

Service function for the ADAU1361 board implementer.

Send a list of command to ADAU1361. All commands has 3 bytes length. That mean, after two byte register address, only 1 byte data pay load is allowed. Commadns are sent by I2C

**7.1.3.5** `virtual void murasaki::Adau1361::SetAuxInputGain ( float left_gain, float right_gain, bool mute = false )` [protected], [virtual]

#### Parameters

<i>left_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>right_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>mute</i>	set true to mute

Other input lines are not killed. To kill it, user have to mute them explicitly.

**7.1.3.6** `virtual void murasaki::Adau1361::SetGain ( murasaki::CodecChannel channel, float left_gain, float right_gain )` [virtual]

#### Parameters

<i>channel</i>	CODEC input output channels like line-in, line-out, aux-in, headphone-out
<i>left_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>right_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.

Implements [murasaki::AudioCodecStrategy](#).

**7.1.3.7** `virtual void murasaki::Adau1361::SetHpOutputGain ( float left_gain, float right_gain, bool mute = false )` [protected], [virtual]

#### Parameters

<i>left_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>right_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>mute</i>	set true to mute

Other out line like line in are not killed. To kill it, user have to mute them explicitly.

**7.1.3.8** `virtual void murasaki::Adau1361::SetLineInputGain ( float left_gain, float right_gain, bool mute = false )` [protected], [virtual]

## Parameters

<i>left_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>right_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>mute</i>	set true to mute

As same as start(), this gain control function uses the single-end negative input only. Other input signal of the line in like positive signal or diff signal are killed.

Other input line like aux are not killed. To kill it, user have to mute them explicitly.

**7.1.3.9** `virtual void murasaki::Adau1361::SetLineOutputGain ( float left_gain, float right_gain, bool mute = false )`  
`[protected], [virtual]`

## Parameters

<i>left_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>right_gain</i>	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
<i>mute</i>	set true to mute

Other output lines are not killed. To kill it, user have to mute them explicitly.

**7.1.3.10** `virtual void murasaki::Adau1361::Start ( void )` `[virtual]`

This method starts the ADAU1361 AD/DA conversion and I2S communication.

The line in is configured to use the Single-End negative input. This is funny but ADAU1361 datasheet specifies to do it. The positive in and diff in are killed. All biases are set as "normal".

The CODEC is configured as master mode. That mean, bclk and WS are given from ADAU1361 to the micro processor.

At initial state, ADAU1361 is set as :

- All input and output channels are set as 0.0dB and muted.

Implements [murasaki::AudioCodecStrategy](#).

**7.1.3.11** `virtual void murasaki::Adau1361::WaitPllLock ( void )` `[protected], [virtual]`

Service function for the ADAu1361 board implementer.

Read the PLL status and repeat it until the PLL locks.

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

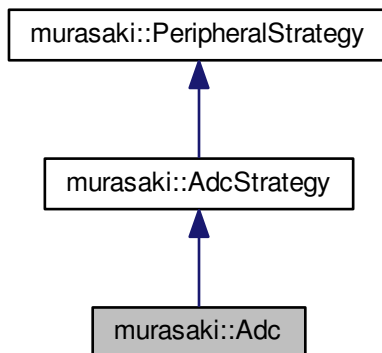
- [adau1361.hpp](#)

## 7.2 murasaki::Adc Class Reference

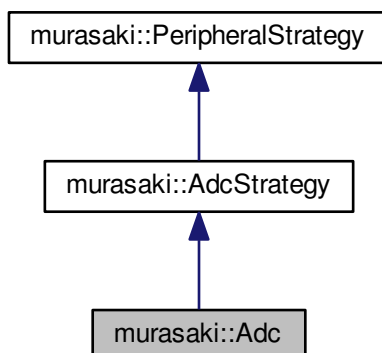
STM32 dedicated ADC class.

```
#include <adc.hpp>
```

Inheritance diagram for murasaki::Adc:



Collaboration diagram for murasaki::Adc:



### Public Member Functions

- [Adc](#) (ADC\_HandleTypeDef \*peripheral)  
*Constructor.*
- virtual [~Adc](#) ()  
*destructor*



- virtual void [SetSampleClock](#) (unsigned int channel, unsigned int clocks)  
*Set the sample clocks for specified channel.*
- virtual [murasaki::AdcStatus Convert](#) (const unsigned int channel, float \*value, unsigned int size=1)  
*Convert the analog input through the given channel by STM32 internal ADC.*
- virtual bool [ConversionCompleteCallback](#) (void \*ptr)  
*Callback function for the interrupt handler.*
- virtual bool [HandleError](#) (void \*ptr)  
*Handling error report of device.*

### Protected Member Functions

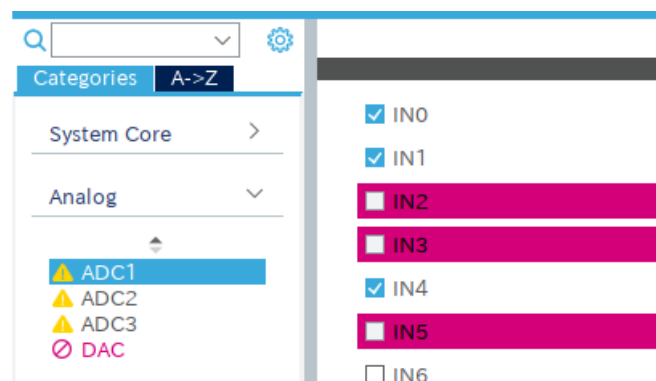
- virtual void \* [GetPeripheralHandle](#) ()  
*pass the raw peripheral handler*

#### 7.2.1 Detailed Description

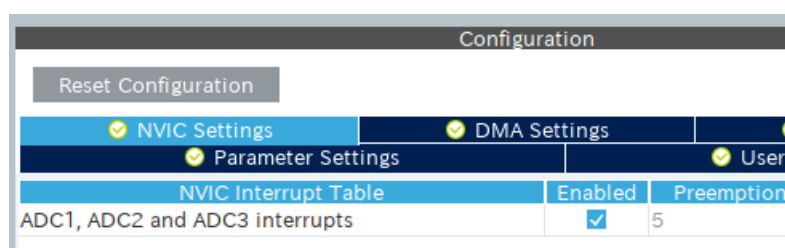
This is a single conversion ADC class. That mean, Only one specified channel will be converted at onces and then, return. Optionally, The sampling clock duration can be set for each channels.

### Configuration

To configure an analog input, select an ADC to use, and check the ADC input line. Note that even checking multiple lines, you can convert only one line at once.



Then, enable the NVIC interrupt line for the ADC.



## Creating a peripheral object

To create an ADC object, use new operator to the [murasaki::Adc](#) class. The parameter is the handle of that ADC. The handle will be generated by CubeIDE around the top of the [main.c](#).

```
murasaki::platform.adc = new murasaki::Adc(&hadc1);
MURASAKI_ASSERT(nullptr != murasaki::platform.adc)
```

## Interrupt Handling

Interrupt is handled automatically. Programmer doesn't need to care.

## ADC data reading

To read the data, use [Adc::Convert\(\)](#) member function.

```
murasaki::platform.adc->Convert(ADC_CHANNEL_TEMPSENSOR, &value);
```

## 7.2.2 Constructor & Destructor Documentation

### 7.2.2.1 murasaki::Adc::Adc ( ADC\_HandleTypeDef \* *peripheral* )

#### Parameters

<i>peripheral</i>	handle of the ADC device in STM32.
-------------------	------------------------------------

## 7.2.3 Member Function Documentation

### 7.2.3.1 bool murasaki::Adc::ConversionCompleteCallback ( void \* *ptr* ) [virtual]

#### Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a peripheral control
------------	---

#### Returns

true: *ptr* matches with *peripheral* and handle the call back. false : doesn't match.

A call back to notify the end of conversion. The definition of calling timing is depend on the implementation. This function must be called from [HAL\\_ADC\\_ConvCpltCallback\(\)](#).

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
    if (murasaki::platform.adc->ConversionCompleteCallback(hadc))
        return;
}
```

Implements [murasaki::AdcStrategy](#).

### 7.2.3.2 murasaki::AdcStatus murasaki::Adc::Convert ( const unsigned int *channel*, float \* *value*, unsigned int *size* = 1 ) [virtual]

## Parameters

<i>channel</i>	Specify the ADC channel. Use ADC_CHANNEL_* value.
<i>value</i>	Pointer to the variable to receive the data. Data is normalized by range [-1,1)
<i>size</i>	Must be 1. Otherwise, assertion failed.

## Returns

Status. murasaki::kasOk is return if success.

Convert the analog signal through the given channel to digital.

The processing is synchronous and blocking. During certain channel is converted, other channels are kept waited.

Implements [murasaki::AdcStrategy](#).

**7.2.3.3** `void * murasaki::Adc::GetPeripheralHandle ( )` [protected],[virtual]

## Returns

pointer to the raw peripheral handler hidden in a class.

Implements [murasaki::PeripheralStrategy](#).

**7.2.3.4** `bool murasaki::Adc::HandleError ( void * ptr )` [virtual]

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a device control
------------	---

## Returns

true if ptr matches with device and handle the error. false if ptr doesn't match A member function to detect error.

This must be called from [HAL\\_ADC\\_ErrorCallback\(\)](#)

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
    if (murasaki::platform.adc->ConversionCompleteCallback(hadc))
        return;
}
```

Implements [murasaki::AdcStrategy](#).

**7.2.3.5** `void murasaki::Adc::SetSampleClock ( unsigned int channel, unsigned int clocks )` [virtual]

## Parameters

<i>channel</i>	Specify the ADC channel. Use ADC_CHANNEL_* value.
<i>clocks</i>	How many clocks is applied to sample. Use ADC_SAMPLETIME_*CYCLES value

Set the dedicated sample clock to the specific ADC channel. The higher impedance signal source may need longer clocks.

The channel parameter is restricted as `ADC_CHANNEL_*` value. Otherwise, HAL may fail the assertion inside `Convert()` member function.

The clocks parameter is restricted as `ADC_SAMPLETIME_*CYCLES`. Otherwise, HAL may fail the assertion inside `Convert()` member function.

Up to 32 channels data can be stored. If more than 32 channels are stored, assertion failed.

Implements `murasaki::AdcStrategy`.

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

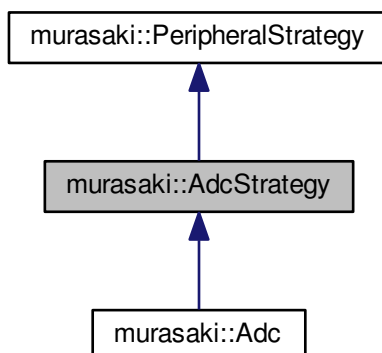
- [adc.hpp](#)
- [adc.cpp](#)

### 7.3 `murasaki::AdcStrategy` Class Reference

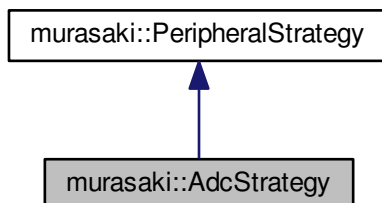
Synchronized, blocking ADC converter Strategy.

```
#include <adcstrategy.hpp>
```

Inheritance diagram for `murasaki::AdcStrategy`:



Collaboration diagram for `murasaki::AdcStrategy`:



## Public Member Functions

- [AdcStrategy](#) ()  
*constructor*
- virtual void [SetSampleClock](#) (unsigned int channel, unsigned int clocks)=0  
*Set the sample clocks for specified channel.*
- virtual [murasaki::AdcStatus Convert](#) (unsigned int channel, float \*value, unsigned int size=1)=0  
*Convert the analog input through the given channel.*
- virtual bool [ConversionCompleteCallback](#) (void \*ptr)=0  
*Call this function when the ADC conversion is done.*
- virtual bool [HandleError](#) (void \*ptr)=0  
*Handling error report of device.*

## Additional Inherited Members

## 7.3.1 Member Function Documentation

7.3.1.1 virtual bool [murasaki::AdcStrategy::ConversionCompleteCallback](#) ( void \* *ptr* ) [pure virtual]

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a peripheral control
------------	---

## Returns

true: ptr matches with peripheral and handle the call back. false : doesn't match.

A call back to notify the end of conversion. The definition of calling timing is depend on the implementation. This is called from an DMA ISR.

Typically, an implementation may check whether the given ptr parameter matches its own device, and if matched, handle it and return true. If it doesn't match, just return false.

Implemented in [murasaki::Adc](#).

7.3.1.2 virtual [murasaki::AdcStatus](#) [murasaki::AdcStrategy::Convert](#) ( unsigned int *channel*, float \* *value*, unsigned int *size* = 1 ) [pure virtual]

## Parameters

<i>channel</i>	Specify the ADC channel.
<i>value</i>	Pointer to the variable to receive the data. Data is normalized by range [-1,1)
<i>size</i>	Length of array if the value is array. Must be 1 if the value is a scalar.

## Returns

Implemented in [murasaki::Adc](#).

7.3.1.3 `virtual bool murasaki::AdcStrategy::HandleError ( void * ptr )` [pure virtual]

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a device control
------------	---

## Returns

true if ptr matches with device and handle the error. false if ptr doesn't match A member function to detect error.

The error handling is depend on the implementation.

Implemented in [murasaki::Adc](#).

**7.3.1.4** `virtual void murasaki::AdcStrategy::SetSampleClock ( unsigned int channel, unsigned int clocks )` [pure virtual]

## Parameters

<i>channel</i>	Specify the ADC channel.
<i>clocks</i>	How many clocks is applied to sample.

Optional implementation dependent setting.

Implemented in [murasaki::Adc](#).

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

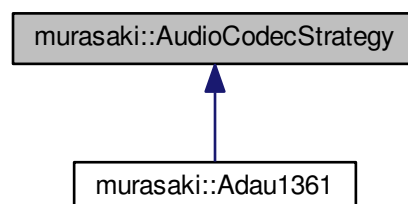
- [adcstrategy.hpp](#)

## 7.4 murasaki::AudioCodecStrategy Class Reference

abstract audio CODEC controller.

```
#include <audiocodecstrategy.hpp>
```

Inheritance diagram for murasaki::AudioCodecStrategy:



## Public Member Functions

- [AudioCodecStrategy](#) (unsigned int fs)  
*constructor.*
- virtual void [Start](#) (void)=0  
*Actual initializer.*
- virtual void [SetGain](#) ([murasaki::CodecChannel](#) channel, float left\_gain, float right\_gain)=0  
*Set channel gain.*
- virtual void [Mute](#) ([murasaki::CodecChannel](#) channel, bool mute=true)=0  
*Mute the specific channel.*
- virtual void [SendCommand](#) (const uint8\_t command[], int size)=0  
*send one command to CODEC*

## Protected Attributes

- unsigned int [fs\\_](#)  
*Sampling Frequency [Hz].*

### 7.4.1 Detailed Description

This class is template for all CODEC classes

### 7.4.2 Constructor & Destructor Documentation

#### 7.4.2.1 [murasaki::AudioCodecStrategy::AudioCodecStrategy](#) ( unsigned int fs ) `[inline]`

##### Parameters

<i>fs</i>	Sampling frequency[Hz].
-----------	-------------------------

initialize the internal variables.

### 7.4.3 Member Function Documentation

#### 7.4.3.1 `virtual void murasaki::AudioCodecStrategy::Mute ( murasaki::CodecChannel channel, bool mute = true )` `[pure virtual]`

##### Parameters

<i>channel</i>	Channel to mute on / off
<i>mute</i>	On if true, off if false.

Implemented in [murasaki::Adau1361](#).

#### 7.4.3.2 `virtual void murasaki::AudioCodecStrategy::SendCommand ( const uint8_t command[], int size )` `[pure virtual]`



## Parameters

<i>command</i>	command data array.
<i>size</i>	command length by [byte].

Implemented in [murasaki::Adau1361](#).

**7.4.3.3** `virtual void murasaki::AudioCodecStrategy::SetGain ( murasaki::CodecChannel channel, float left_gain, float right_gain )` [pure virtual]

## Parameters

<i>channel</i>	
<i>left_gain</i>	[dB]
<i>right_gain</i>	[dB]

Implemented in [murasaki::Adau1361](#).

**7.4.3.4** `virtual void murasaki::AudioCodecStrategy::Start ( void )` [pure virtual]

Initialize the CODEC and start the conversion process.

Implemented in [murasaki::Adau1361](#).

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

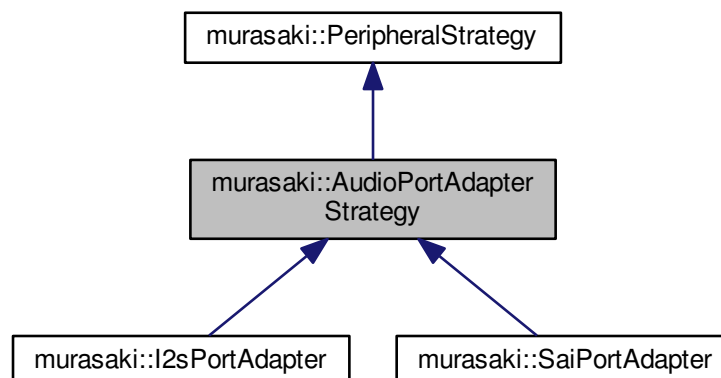
- [audiocodecstrategy.hpp](#)

## 7.5 murasaki::AudioPortAdapterStrategy Class Reference

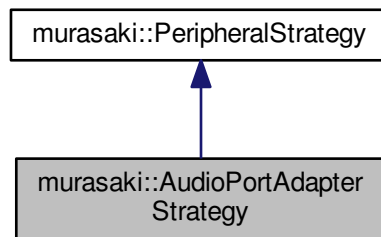
Strategy of the audio device adaptor..

```
#include <audioportadapterstrategy.hpp>
```

Inheritance diagram for murasaki::AudioPortAdapterStrategy:



Collaboration diagram for `murasaki::AudioPortAdapterStrategy`:



### Public Member Functions

- virtual void [StartTransferTx](#) (uint8\_t \*tx\_buffer, unsigned int channel\_len)=0  
*Kick start routine to start the TX DMA transfer.*
- virtual void [StartTransferRx](#) (uint8\_t \*rx\_buffer, unsigned int channel\_len)=0  
*Kick start routine to start the RX DMA transfer.*
- virtual unsigned int [GetNumberOfDMAPhase](#) ()=0  
*Return how many DMA phase is implemented.*
- virtual unsigned int [GetNumberOfChannelsTx](#) ()=0  
*Return how many channels are in the transfer.*
- virtual unsigned int [GetSampleShiftSizeTx](#) ()=0  
*Return the bit count to shift the 1.31 fixed point data to the DMA required format.*
- virtual unsigned int [GetSampleWordSizeTx](#) ()=0  
*Return the size of the one sample on memory for Tx channel.*
- virtual unsigned int [GetNumberOfChannelsRx](#) ()=0  
*Return how many channels are in the transfer.*
- virtual unsigned int [GetSampleShiftSizeRx](#) ()=0  
*Return the bit count to shift the DMA format to the left aligned 1.31 format. .*
- virtual unsigned int [GetSampleWordSizeRx](#) ()=0  
*Return the size of the one sample on memory for Rx channel.*
- virtual unsigned int [DetectPhase](#) (unsigned int phase)  
*DMA phase detector.*
- virtual bool [HandleError](#) (void \*ptr)=0  
*Handling error report of device.*
- virtual bool [Match](#) (void \*peripheral\_handle)=0
- virtual void \* [GetPeripheralHandle](#) ()=0  
*pass the raw peripheral handler*
- virtual bool [IsInt16SwapRequired](#) ()=0  
*Display half word swap is required. .*

### Additional Inherited Members

#### 7.5.1 Detailed Description

Template class of the audio device adaptor.

## 7.5.2 Member Function Documentation

7.5.2.1 `virtual unsigned int murasaki::AudioPortAdapterStrategy::DetectPhase ( unsigned int phase ) [inline], [virtual]`

## Parameters

<i>phase</i>	RX DMA phase : 0, 1, ...
--------------	--------------------------

## Returns

By default, returns phase parameter.

If the DMA interrupt doesn't have the explicit phase information, need to override to detect it inside this function.

By default, this function assumes the DMA phase is given though the interrupt handler. So, just pass the input parameter as return value.

7.5.2.2 `virtual unsigned int murasaki::AudioPortAdapterStrategy::GetNumberOfChannelsRx ( ) [pure virtual]`

## Returns

1 for Mono, 2 for stereo, 3... for multi-channel.

Implemented in [murasaki::I2sPortAdapter](#), and [murasaki::SaiPortAdapter](#).

7.5.2.3 `virtual unsigned int murasaki::AudioPortAdapterStrategy::GetNumberOfChannelsTx ( ) [pure virtual]`

## Returns

1 for Mono, 2 for stereo, 3... for multi-channel.

Implemented in [murasaki::SaiPortAdapter](#), and [murasaki::I2sPortAdapter](#).

7.5.2.4 `virtual unsigned int murasaki::AudioPortAdapterStrategy::GetNumberOfDMAPhase ( ) [pure virtual]`

## Returns

2 for Double buffer, 3 for Tripple buffer.

Implemented in [murasaki::SaiPortAdapter](#), and [murasaki::I2sPortAdapter](#).

7.5.2.5 `virtual void* murasaki::AudioPortAdapterStrategy::GetPeripheralHandle ( ) [pure virtual]`

## Returns

pointer to the raw peripheral handler hidden in a class.

Implements [murasaki::PeripheralStrategy](#).

Implemented in [murasaki::I2sPortAdapter](#), and [murasaki::SaiPortAdapter](#).

**7.5.2.6** `virtual unsigned int murasaki::AudioPortAdapterStrategy::GetSampleShiftSizeRx ( ) [pure virtual]`

#### Returns

0..32. The unit is [bit]

This is needed because of the variation of the DMA format.

Let's assume the 24bit data I2S format. Some peripheral places the data as right aligned in 32bit DMA data ( as integer ), some peripheral places the data as left aligned in 32bit DMA data ( as fixed point ).

This kind of the mismatch will be aligned by audio frame work. This member function returns how many bits have to be shifted from the DMA format to the left aligned data.

If peripheral DMA uses the left aligned format, this function should return 0.

Implemented in [murasaki::I2sPortAdapter](#), and [murasaki::SaiPortAdapter](#).

**7.5.2.7** `virtual unsigned int murasaki::AudioPortAdapterStrategy::GetSampleShiftSizeTx ( ) [pure virtual]`

#### Returns

0..32. The unit is [bit]

This is needed because of the variation of the DMA format.

Let's assume the 24bit data I2S format. Some peripheral places the data as right aligned in 32bit DMA data ( as integer ), some peripheral places the data as left aligned in 32bit DMA data ( as fixed point ).

This kind of the mismatch will be aligned by audio frame work. This member function returns how many bits have to be shifted from left aligned data to the DMA required format..

If peripheral requires the left aligned format, this function should return 0.

Implemented in [murasaki::SaiPortAdapter](#), and [murasaki::I2sPortAdapter](#).

**7.5.2.8** `virtual unsigned int murasaki::AudioPortAdapterStrategy::GetSampleWordSizeRx ( ) [pure virtual]`

#### Returns

2 or 4. The unit is [Byte]

This function returns the size of the word which should be allocated on the memory.

Implemented in [murasaki::I2sPortAdapter](#), and [murasaki::SaiPortAdapter](#).

**7.5.2.9** `virtual unsigned int murasaki::AudioPortAdapterStrategy::GetSampleWordSizeTx ( ) [pure virtual]`

#### Returns

2 or 4. The unit is [Byte]

This function returns the size of the word which should be allocated on the memory.

Implemented in [murasaki::I2sPortAdapter](#), and [murasaki::SaiPortAdapter](#).

**7.5.2.10** `virtual bool murasaki::AudioPortAdapterStrategy::HandleError ( void * ptr ) [pure virtual]`

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a device control
------------	---

## Returns

true if ptr matches with device and handle the error. false if ptr doesn't match

A member function to detect error.

Note, we assume once this error call back is called, we can't recover.

Implemented in [murasaki::I2sPortAdapter](#), and [murasaki::SaiPortAdapter](#).

#### 7.5.2.11 virtual bool murasaki::AudioPortAdapterStrategy::IsInt16SwapRequired ( ) [pure virtual]

## Returns

true : half word swap required. false : half word swap is not required.

Display whether the half word (int16\_t) swap is required or not.

Certain architecture requires to swap the upper half word and lower half word inside a word (int32\_t). In case this is required before copying to TX DMA buffer or after copying from RX DMA buffer, return true. Otherwise, return false.

The return value doesn't affect to the endian inside half word. This display will be ignored if the audio sample size is half word (int16\_t) or byte (int8\_t).

Implemented in [murasaki::I2sPortAdapter](#), and [murasaki::SaiPortAdapter](#).

#### 7.5.2.12 virtual bool murasaki::AudioPortAdapterStrategy::Match ( void \* *peripheral\_handle* ) [pure virtual]

Check if peripheral handle matched with given handle.

## Parameters

<i>peripheral_handle</i>	
--------------------------	--

## Returns

true if match, false if not match.

Reimplemented from [murasaki::PeripheralStrategy](#).

Implemented in [murasaki::I2sPortAdapter](#), and [murasaki::SaiPortAdapter](#).

#### 7.5.2.13 virtual void murasaki::AudioPortAdapterStrategy::StartTransferRx ( uint8\_t \* *rx\_buffer*, unsigned int *channel\_len* ) [pure virtual]

This routine must be implemented by the derived class. The task of this routine is to kick the first DMA transfer. In this class, we assume DMA continuously transfer on the circular buffer once after it starts.

Implemented in [murasaki::SaiPortAdapter](#), and [murasaki::I2sPortAdapter](#).

**7.5.2.14** `virtual void murasaki::AudioPortAdapterStrategy::StartTransferTx ( uint8_t * tx_buffer, unsigned int channel_len )`  
[pure virtual]

This routine must be implemented by the derived class. The task of this routine is to kick the first DMA transfer. In this class, we assume DMA continuously transfer on the circular buffer once after it starts.

Implemented in [murasaki::SaiPortAdapter](#), and [murasaki::I2sPortAdapter](#).

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

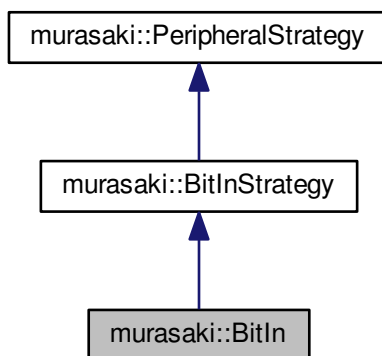
- [audioportadapterstrategy.hpp](#)

## 7.6 murasaki::BitIn Class Reference

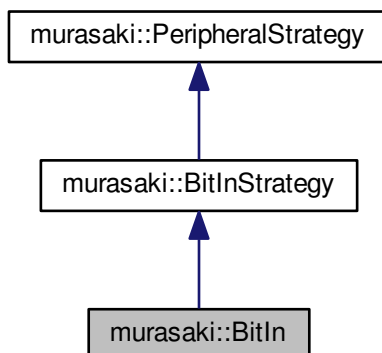
General purpose bit input.

```
#include <bitin.hpp>
```

Inheritance diagram for murasaki::BitIn:



Collaboration diagram for murasaki::BitIn:



## Public Member Functions

- [BitIn](#) (GPIO\_TypeDef \*port, uint16\_t pin)  
*Constructor.*
- virtual unsigned int [Get](#) (void)  
*Get a status of the input pin.*
- virtual void \* [GetPeripheralHandle](#) ()  
*pass the raw peripheral handler*

## Additional Inherited Members

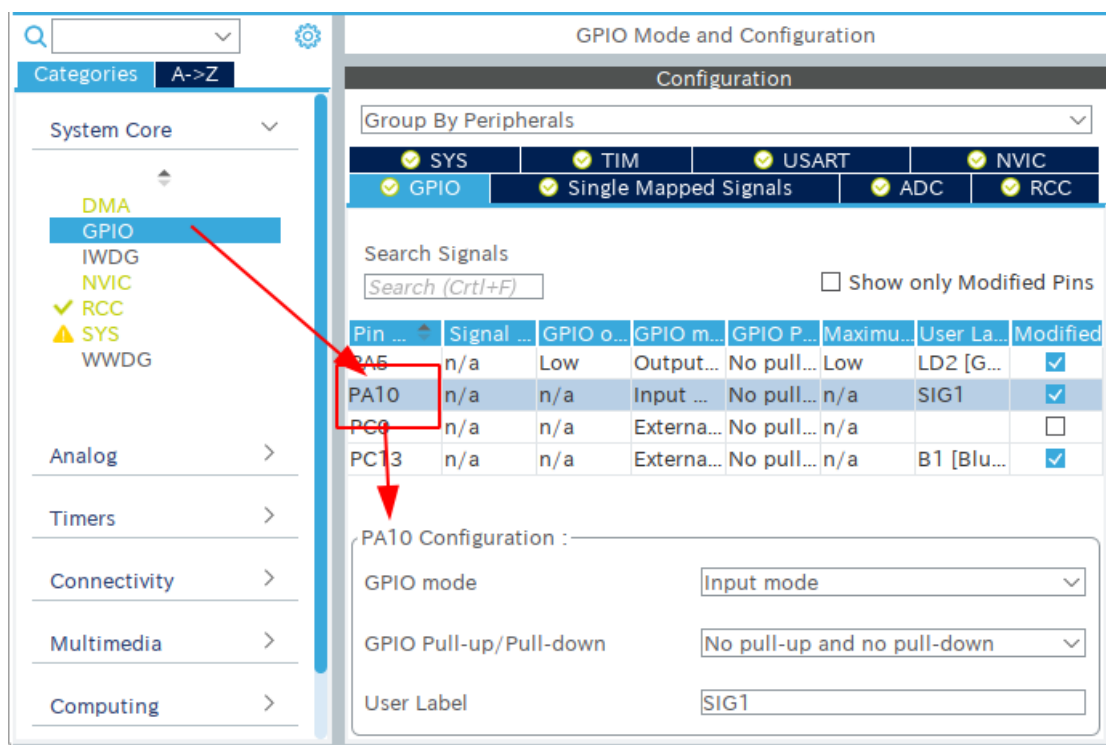
## 7.6.1 Detailed Description

The [BitIn](#) class is the wrapper of the GPIO controller. Programmer can read the state of a signal by using [BitIn](#) class.

## Configuration

To configure a bit input, open the Device Configuration Tool of the CubeIDE. Make appropriate pins as GPIO Input pin. Then, chose GPIO section from the left pane, select the input pin and configure.

The string in the User Label filed can be used in the application program. In this example, that is "SIG1"



## Creating a peripheral object

To use the [BitIn](#) class, create an instance with GPIO\_TypeDef \* type pointer. For example, to create an instance for a switch peripheral:

```
my_signal = new murasaki::BitIn(SIG1_GPIO_port, SIG1_pin);
```

Where "SIG1" is the user label defined in the configuration. The \*\*\*\_GPIO\_port and \*\*\*\_pin are generated by CubeIDE automatically from the user label in the configuration.

## Reading Data

To read data, use the Get member function.

```
unsigned int state = my_signal.Get();
```

The return value is 1 for "H" input, 0 for "L" input.

### 7.6.2 Constructor & Destructor Documentation

#### 7.6.2.1 `murasaki::BitIn ( GPIO_TypeDef * port, uint16_t pin )`

##### Parameters

<i>port</i>	Pointer to the port struct.
<i>pin</i>	Number of the pin to input.

### 7.6.3 Member Function Documentation

#### 7.6.3.1 `unsigned int murasaki::BitIn::Get ( void ) [virtual]`

##### Returns

1 or 0 as output state.

Mean of "1" or "0" is system dependent.

Usually, these represent "H" or "L" output state, respectively.

Implements [murasaki::BitInStrategy](#).

#### 7.6.3.2 `void * murasaki::BitIn::GetPeripheralHandle ( ) [virtual]`

##### Returns

pointer to the [GPIO\\_type](#) variable hidden in a class.

Implements [murasaki::PeripheralStrategy](#).

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

- [bitin.hpp](#)
- [bitin.cpp](#)

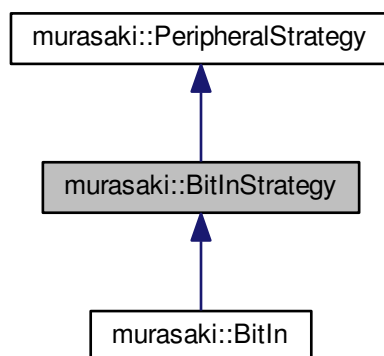


## 7.7 murasaki::BitInStrategy Class Reference

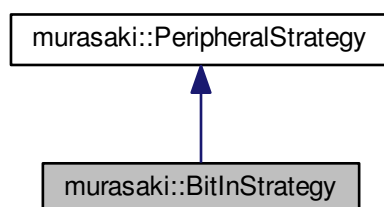
Definition of the root class of bit input.

```
#include <bitinstrategy.hpp>
```

Inheritance diagram for murasaki::BitInStrategy:



Collaboration diagram for murasaki::BitInStrategy:



### Public Member Functions

- virtual unsigned int `Get` (void)=0  
*Get a status of the input pin.*

### Additional Inherited Members

#### 7.7.1 Detailed Description

A prototype of the general purpose bit input class

## 7.7.2 Member Function Documentation

### 7.7.2.1 `virtual unsigned int murasaki::BitInStrategy::Get ( void ) [pure virtual]`

#### Returns

1 or 0 as input state.

The mean of "1" or "0" is system dependent.

Usually, these represent "H" or "L" input state, respectively.

Implemented in [murasaki::BitIn](#).

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

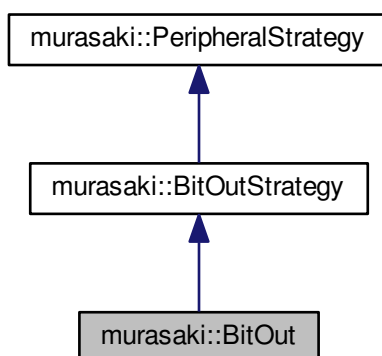
- [bitinstrategy.hpp](#)

## 7.8 murasaki::BitOut Class Reference

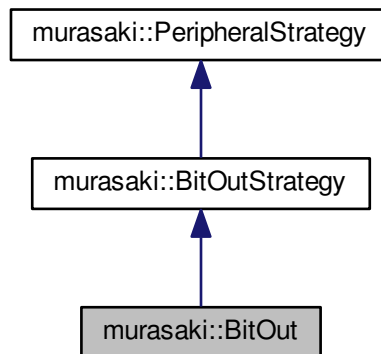
General purpose bit output.

```
#include <bitout.hpp>
```

Inheritance diagram for murasaki::BitOut:



Collaboration diagram for murasaki::BitOut:



#### Public Member Functions

- `BitOut` (`GPIO_TypeDef *port, uint16_t pin`)  
*Constructor.*
- virtual void `Set` (`unsigned int state=1`)  
*Set a status of the output pin.*
- virtual unsigned int `Get` (`void`)  
*Get a status of the output pin.*
- virtual void \* `GetPeripheralHandle` ()  
*pass the raw peripheral handler*

#### Additional Inherited Members

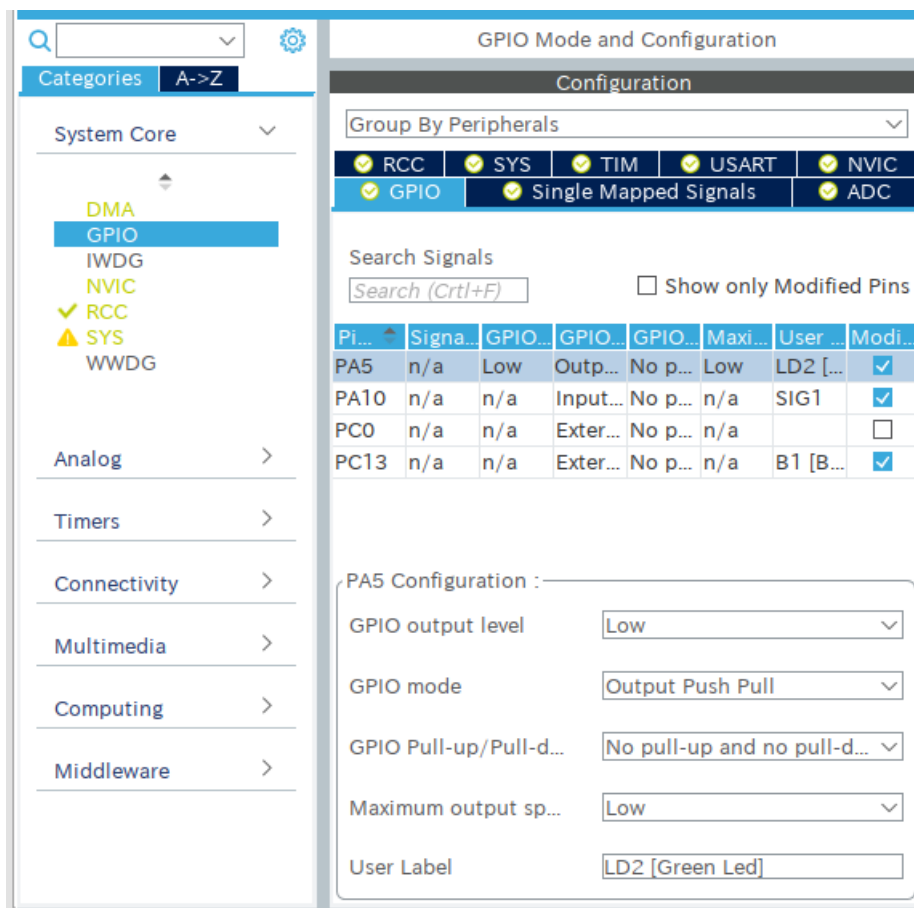
##### 7.8.1 Detailed Description

The `BitOut` class is the wrapper of the GPIO controller. Programmer can change the output pin state by using the `BitOut` class.

#### Configuration

To configure a bit output, open the Device Configuration Tool of the CubeIDE. Make appropriate pins as GPIO Output pin. Then, chose GPIO section from the left pane, select the output pin and configure.

The string in the User Label filed can be used in the application program. In this example, that is "LD2"



### Creating a peripheral object

To use the `BitOut` class, create an instance with `GPIO_TypeDef * type` pointer. For example, to create an instance for LED control:

```
my_led = new murasaki::BitOut (LD2_GPIO_port, LD2_pin);
```

Where "LD2" is the user label defined in the configuration. The `***_GPIO_port` and `***_pin` are generated by CubeIDE automatically from the user label in the configuration.

### Writing Data

To write data, use the `Set` member function.

```
my_led.Set();
```

The output state is set to "H".

You can call `Clear()` member function to set the output to "L".

```
my_led.Cler();
```

Or you can call `Toggle()` member function to flip the output signal.

```
my_led.Toggle();
```

If the previous output was "L", the output change to "H". If the previous output was "H", the output change to "L".

## 7.8.2 Constructor & Destructor Documentation

### 7.8.2.1 `murasaki::BitOut::BitOut ( GPIO_TypeDef * port, uint16_t pin )`

## Parameters

<i>port</i>	Pinter to the port strict.
<i>pin</i>	Number of the pin to output.

## 7.8.3 Member Function Documentation

## 7.8.3.1 unsigned int murasaki::BitOut::Get ( void ) [virtual]

## Returns

1 or 0 as output state.

The mean of "1" or "0" is system dependent.

Usually, these represent "H" or "L" output state, respectively.

Implements [murasaki::BitOutStrategy](#).

## 7.8.3.2 void \* murasaki::BitOut::GetPeripheralHandle ( ) [virtual]

## Returns

pointer to the [GPIO\\_type](#) variable hidden in a class.

Implements [murasaki::PeripheralStrategy](#).

## 7.8.3.3 void murasaki::BitOut::Set ( unsigned int state = 1 ) [virtual]

## Parameters

<i>state</i>	Set "H" if the value is none zero, vice versa.
--------------	--

Implements [murasaki::BitOutStrategy](#).

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

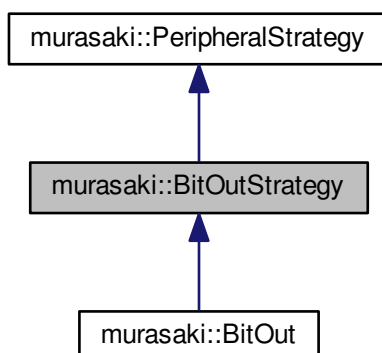
- [bitout.hpp](#)
- [bitout.cpp](#)

## 7.9 murasaki::BitOutStrategy Class Reference

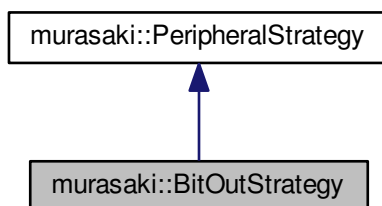
Definition of the root class of bit output.

```
#include <bitoutstrategy.hpp>
```

Inheritance diagram for `murasaki::BitOutStrategy`:



Collaboration diagram for `murasaki::BitOutStrategy`:



### Public Member Functions

- virtual void `Set` (unsigned int state=1)=0  
*Set a status of the output pin.*
- virtual void `Clear` (void)  
*Clear the status of the output pin as "L".*
- virtual unsigned int `Get` (void)=0  
*Get a status of the output pin.*
- virtual void `Toggle` (void)  
*Set "L" if current status is "H", vice versa.*

### Additional Inherited Members

#### 7.9.1 Detailed Description

A prototype of the general purpose bit out class

## 7.9.2 Member Function Documentation

7.9.2.1 `virtual void murasaki::BitOutStrategy::Clear ( void ) [inline],[virtual]`

Usually successor class doesn't overload this member function.

7.9.2.2 `virtual unsigned int murasaki::BitOutStrategy::Get ( void ) [pure virtual]`

## Returns

1 or 0 as output state.

The mean of "1" or "0" is system dependent.

Usually, these represent "H" or "L" output state, respectively.

Implemented in [murasaki::BitOut](#).

7.9.2.3 `virtual void murasaki::BitOutStrategy::Set ( unsigned int state = 1 ) [pure virtual]`

## Parameters

<i>state</i>	Set "H" if the value is none zero, vice versa.
--------------	--

Implemented in [murasaki::BitOut](#).

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

- [bitoutstrategy.hpp](#)

7.10 `murasaki::CallbackRepositorySingleton` Class Reference

Repository of the interrupt handling objects.

```
#include <callbackrepositorysingleton.hpp>
```

## Public Member Functions

- void [AddPeripheralObject](#) ([murasaki::PeripheralStrategy](#) \*peripheral\_object)  
*Add the given object to the internal list of the peripheral.*
- [murasaki::PeripheralStrategy](#) \* [GetPeripheralObject](#) (void \*peripheral\_handle)  
*Search an interrupt responding object.*

## Static Public Member Functions

- static [CallbackRepositorySingleton](#) \* [GetInstance](#) ()  
*Return a singleton instance.*

### 7.10.1 Detailed Description

A Singleton design repository. This is Murasaki internal only. Do not use in the application.

This class object stores the list of the interrupt callback responding objects. The registration is done in the client classe's construction phase. So, it is invisible from the application programmer.

For each interrupt, the callback routines call the [GetPeripheralObject\(\)](#) with given peripheral handler. The [GetPeripheralHandler\(\)](#) routine search for the list whether responding object exists or not. If exist, return that object.

The EXTI is not supported by this class.

This class is not thread safe.

### 7.10.2 Member Function Documentation

#### 7.10.2.1 `void murasaki::CallbackRepositorySingleton::AddPeripheralObject ( murasaki::PeripheralStrategy * peripheral_object )`

##### Parameters

<code>peripheral_object</code>	Add this object to the list.
--------------------------------	------------------------------

Store the given object to the internal list. This member function is called from only inside [PeripheralStrategy](#) or its sub classes.

#### 7.10.2.2 `CallbackRepositorySingleton * murasaki::CallbackRepositorySingleton::GetInstance ( ) [static]`

##### Returns

Instance of this class. For each time of calling, this function returns identical pointer.

#### 7.10.2.3 `murasaki::PeripheralStrategy * murasaki::CallbackRepositorySingleton::GetPeripheralObject ( void * peripheral_handle )`

##### Parameters

<code>peripheral_handle</code>	Search the object which has this handle.
--------------------------------	--

##### Returns

Matched object pointer.

If the object is not found, Assertion failed.

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

- [callbackrepositorysingleton.hpp](#)
- [callbackrepositorysingleton.cpp](#)



## 7.11 murasaki::CriticalSection Class Reference

A critical section for task context.

```
#include <criticalsection.hpp>
```

### Public Member Functions

- [CriticalSection](#) ()  
*Constructor. Creating semaphore internally.*
- virtual [~CriticalSection](#) ()  
*Destructor. Deleting semaphore internally.*
- void [Enter](#) ()  
*Entering critical section.*
- void [Leave](#) ()  
*Leaving critical section.*

### 7.11.1 Detailed Description

The critical section prevent other task to preempt that critical section. So, a task can modify the shared variable safely inside critical section.

This class provide a critical section for the task context only. This critical section is not protected from the ISR.

The critical section have to start by [CriticalSection::Enter\(\)](#) and quit by [CriticalSection::Leave\(\)](#).

### 7.11.2 Member Function Documentation

#### 7.11.2.1 void murasaki::CriticalSection::Enter ( )

Entering critical section in task context. No other task can preemptive the task inside critical section.

#### 7.11.2.2 void murasaki::CriticalSection::Leave ( )

All critical section started by [CriticalSection::Enter\(\)](#) have to be quit by this member function.

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

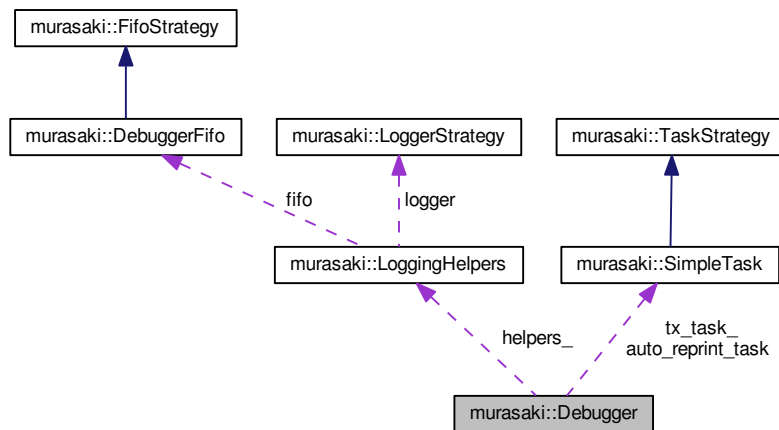
- [criticalsection.hpp](#)
- [criticalsection.cpp](#)

## 7.12 murasaki::Debugger Class Reference

Debug class. Provides printf() style output for both task and ISR context.

```
#include <debugger.hpp>
```

Collaboration diagram for murasaki::Debugger:



### Public Member Functions

- [Debugger](#) ([LoggerStrategy](#) \*logger)  
*Constructor. Create internal variable.*
- virtual [~Debugger](#) ()  
*Destructor. Delete internal variable.*
- void [Printf](#) (const char \*fmt,...)  
*Debug output function.*
- char [GetchFromTask](#) ()  
*Receive one character from serial port.*
- void [RePrint](#) ()  
*Print the old data again.*
- void [AutoRePrint](#) ()  
*Print history automatically.*
- void [DoPostMortem](#) ()  
*Start the post mortem mprocessing. Never return.*

### Protected Attributes

- [murasaki::SimpleTask](#) \*const tx\_task\_  
*Handle to the transmission control task.*
- [murasaki::SimpleTask](#) \* auto\_reprint\_task  
*Handle to the auto reprint task.*
- bool auto\_reprint\_enable\_

- *For protecting from double enabled.*  
char [line\\_](#) [PLATFORM\_CONFIG\_DEBUG\_LINE\_SIZE]  
*FIFO for the snprintf()*
- [murasaki::SyslogSeverity severity\\_](#)  
*Syslog severity threshold.*
- uint32\_t [facility\\_mask\\_](#)  
*Syslog facility filter mask.*

### 7.12.1 Detailed Description

Wrapper class to help the printf debug. The printf() member function can be called from both task context and ISR context.

There are several configurable parameters of this class:

- [PLATFORM\\_CONFIG\\_DEBUG\\_BUFFER\\_SIZE](#)
- [PLATFORM\\_CONFIG\\_DEBUG\\_LINE\\_SIZE](#)
- [PLATFORM\\_CONFIG\\_DEBUG\\_TASK\\_STACK\\_SIZE](#)
- [PLATFORM\\_CONFIG\\_DEBUG\\_TASK\\_PRIORITY](#)
- [PLATFORM\\_CONFIG\\_DEBUG\\_SERIAL\\_TIMEOUT](#)

See [Application Specific Platform](#) as example this class.

### 7.12.2 Constructor & Destructor Documentation

#### 7.12.2.1 `murasaki::Debugger::Debugger ( LoggerStrategy * logger )`

##### Parameters

<i>logger</i>	The pointer to the <a href="#">LoggerStrategy</a> wrapper class variable.
---------------	---

### 7.12.3 Member Function Documentation

#### 7.12.3.1 `void murasaki::Debugger::AutoRePrint ( )`

Once this member function is called, internally new task is created. This new task watches input by [GetchFromTask\(\)](#) and for each input char is received, trigger the [RePrint\(\)](#).

This auto reprint function is irreversible. Once auto reprint is triggered, there is no way to stop the auto reprint. The second call for the AutoHistory may be ignored

This member function have to be called from task context.

### 7.12.3.2 `char murasaki::Debugger::GetcharFromTask ( )`

#### Returns

Received character.

A blocking function which returns received character. The receive is done on the UART which is passed to the constructor.

This is thread safe and task context dedicated function. Never call from ISR.

Be careful, this is synchronous and blocking while the `Debug::Printf()` is asynchronous and non-blocking.

### 7.12.3.3 `void murasaki::Debugger::Printf ( const char * fmt, ... )`

#### Parameters

<i>fmt</i>	Format string
...	optional parameters

The `printf()` compatible member function. This function can be called from both task context and ISR context. This member function internally calls `sprintf()` variant. So, the parameter processing is fully compatible with `printf()`.

The formatted string is stored in the internal circular buffer. And data inside buffer is transmitted through the UART which is passed by constructor. If the buffer is overflowed, this member function stores as possible, and discard the rest of string. That mean, this member function is neither synchronous nor blocking.

This member function is non-blocking, non-asynchronous, thread safe and re-entrant.

At 2018/Jan/14 measurement, 49bytes was used.

### 7.12.3.4 `void murasaki::Debugger::RePrint ( )`

Must call from task context. For each time this member function is called, old data in the buffer is re-sent again.

The data to be re-sent is the one in the data in side circular buffer. Then, the resent size is same as [PLATFORM\\_CONFIG\\_DEBUG\\_BUFFER\\_SIZE](#) .

## 7.12.4 Member Data Documentation

### 7.12.4.1 `uint32_t murasaki::Debugger::facility_mask_ [protected]`

If certain bit is "1", the corresponding Syslog facility is allowed to output. By default the value is 0xFFFFFFFF ( equivalent to `SyslogAllowAllFacilities(0xFFFFFFFF)` )

### 7.12.4.2 `char murasaki::Debugger::line_[PLATFORM_CONFIG_DEBUG_LINE_SIZE] [protected]`

This variable can be local variable of the `printf()` member function. In this case, the implementation of the `printf()` is much easier. In the other hand, each task must has enough depth on its task stack.

Probably, having bigger task for each task doesn't pay, and it may cuase stack overflow bug at the debug or assertion. This is not preferable.

#### 7.12.4.3 `murasaki::SyslogSeverity` `murasaki::Debugger::severity_` [protected]

All severity level lower than this value will be ignored by `Syslog()` function. Note that `murasaki::kseEmergency` is the highest and `murasaki::kseDebug` is the lowest severity.

By default, the severity level threshold is `murasaki::kseError`. That mean, the weaker severity than `kseError` is ignored.

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

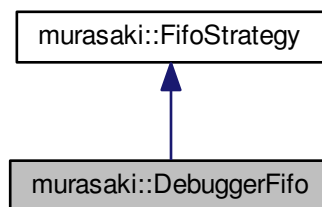
- [debugger.hpp](#)
- [debugger.cpp](#)

### 7.13 `murasaki::DebuggerFifo` Class Reference

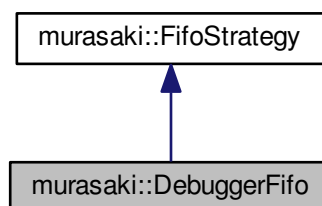
FIFO with thread safe.

```
#include <debuggerfifo.hpp>
```

Inheritance diagram for `murasaki::DebuggerFifo`:



Collaboration diagram for `murasaki::DebuggerFifo`:



## Public Member Functions

- [DebuggerFifo](#) (unsigned int buffer\_size)  
*Create an internal buffer.*
- virtual [~DebuggerFifo](#) ()  
*Delete an internal buffer.*
- virtual void [NotifyData](#) ()  
*Notyify new data is in the buffer, to the receiver task.*
- virtual unsigned int [Get](#) (uint8\_t data[ ], unsigned int size)  
*Get the data from the internal buffer. This is thread safe function. Do not call from ISR.*
- virtual void [ReWind](#) ()  
*Mark all the data inside the internal buffer as "not sent". Thread safe.*
- virtual void [SetPostMortem](#) ()  
*Transit to the post mortem mode.*

## Additional Inherited Members

### 7.13.1 Detailed Description

Non blocking , thread safe FIFO

The Put member function returns with "copied" data count. If the internal buffer is full, it returns without copy data. This is thread safe and ISR/Task bi-modal.

The Get member function returns with "copied" data count and data. If the internal buffer is empty, it returns without copied data.

### 7.13.2 Constructor & Destructor Documentation

#### 7.13.2.1 `murasaki::DebuggerFifo::DebuggerFifo ( unsigned int buffer_size )`

##### Parameters

<code>buffer_size</code>	Size of the internal buffer to be allocated [byte]
--------------------------	--

Allocate an internal buffer with given buffer\_size. The buffer contents is initialized by blank.

### 7.13.3 Member Function Documentation

#### 7.13.3.1 `unsigned int murasaki::DebuggerFifo::Get ( uint8_t data[ ], unsigned int size )` [virtual]

##### Parameters

<code>data</code>	Data buffer to receive from the internal buffer
<code>size</code>	Size of the data parameter.

**Returns**

The count of copied data. 0, if the internal buffer is empty

Reimplemented from [murasaki::FifoStrategy](#).

**7.13.3.2 void murasaki::DebuggerFifo::SetPostMortem ( ) [virtual]**

In this mode, FIFO doesn't sync between the put and get method. Actually, this mode assumes nobody send message by [Put\(\)](#)

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

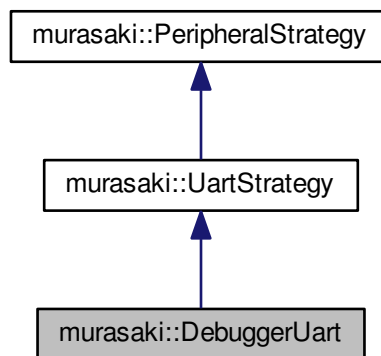
- [debuggerfifo.hpp](#)
- [debuggerfifo.cpp](#)

**7.14 murasaki::DebuggerUart Class Reference**

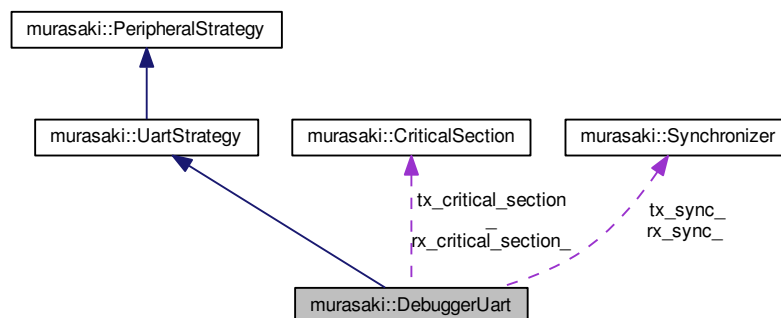
Logging dedicated UART class.

```
#include <debuggeruart.hpp>
```

Inheritance diagram for murasaki::DebuggerUart:



Collaboration diagram for murasaki::DebuggerUart:



## Public Member Functions

- [DebuggerUart](#) (UART\_HandleTypeDef \*uart)  
*Constructor.*
- virtual [~DebuggerUart](#) ()  
*Destructor. Delete internal variables.*
- virtual void [SetHardwareFlowControl](#) (UartHardwareFlowControl control)  
*Set the behavior of the hardware flow control.*
- virtual void [SetSpeed](#) (unsigned int baud\_rate)  
*Set the BAUD rate.*
- virtual [murasaki::UartStatus Transmit](#) (const uint8\_t \*data, unsigned int size, unsigned int timeout\_ms)  
*Transmit raw data through an UART by synchronous mode.*
- virtual [murasaki::UartStatus Receive](#) (uint8\_t \*data, unsigned int count, unsigned int \*transferred\_count, [UartTimeout](#) uart\_timeout, unsigned int timeout\_ms)  
*Receive raw data through an UART by synchronous mode.*
- virtual bool [TransmitCompleteCallback](#) (void \*const ptr)  
*Call back for entire block transfer completion.*
- virtual bool [ReceiveCompleteCallback](#) (void \*const ptr)  
*Call back for entire block transfer completion.*
- virtual bool [HandleError](#) (void \*const ptr)  
*Error handling.*

## Additional Inherited Members

### 7.14.1 Detailed Description

The [Uart](#) class is the wrapper of the UART controller. To use the [DebuggerUart](#) class, make an instance with UART\_HandleTypeDef \* type pointer. For example, to create an instance for the UART3 peripheral :

```
my_uart3 = new murasaki::DebuggerUart(&huart3);
```

Where huart3 is the handle generated by CubeIDE for UART3 peripheral. To use this class, the UART peripheral have to be configured to use the DMA functionality. The baud rate, length and flow control should be configured by the CubeIDE.

The [Uart::Transmit\(\)](#) member function is a synchronous function. A programmer can specify the timeout by timeout\_ms parameter. By default, this parameter is set by kwmsIndefinitely which specifies never time out.

The [Uart::Receive\(\)](#) member function is a synchronous function. A programmer can specify the timeout by timeout\_ms parameter. By default, this parameter is set by kwmsIndefinitely which specifies never time out.

Both methods can be called from only the task context. If these are called in the ISR context, the result is unknown.

### 7.14.2 Constructor & Destructor Documentation

#### 7.14.2.1 murasaki::DebuggerUart::DebuggerUart ( UART\_HandleTypeDef \* uart )



## Parameters

<i>uart</i>	Pointer to a UART control struct. This device have to be configured to use DMA and interrupt for both Tx and Rx.
-------------	--

Store the given uart pointer into the internal variable. This pointer is passed to the STM32Cube HAL UART functions when needed.

## 7.14.3 Member Function Documentation

7.14.3.1 `bool murasaki::DebuggerUart::HandleError ( void *const ptr ) [virtual]`

## Parameters

<i>ptr</i>	Pointer to UART_HandleTypeDef struct.
------------	---------------------------------------

## Returns

true: ptr matches with UART device and handle the error. false : doesn't match.

A handle to print out the error message.

Checks whether handle has error and if there is, print appropriate error. Then trigger an assertion.

Implements [murasaki::UartStrategy](#).

7.14.3.2 `murasaki::UartStatus murasaki::DebuggerUart::Receive ( uint8_t * data, unsigned int count, unsigned int * transfered_count, UartTimeout uart_timeout, unsigned int timeout_ms ) [virtual]`

## Parameters

<i>data</i>	Data buffer to place the received data..
<i>count</i>	The count of the data ( byte ) to be transfered. Must be smaller than 65536
<i>transfered_count</i>	This parameter is ignored.
<i>uart_timeout</i>	This parameter is ignored
<i>timeout_ms</i>	Time out limit by milliseconds.

## Returns

Always returns OK

Receive to given data buffer through an UART device.

The receiving mode is synchronous. That means, function returns when specified number of data has been received, except timeout. Passing [murasaki::kwmsIndefinitely](#) to the parameter timeout\_ms orders not to return until complete receiving. Other value of timeout\_ms parameter specifies the time out by millisecond. If time out happen, function returns false. If not happen, it returns true.

This function is exclusive. Internally this function is guarded by mutex. Then this function is thread safe. This function is forbidden to call from ISR.

Implements [murasaki::UartStrategy](#).

### 7.14.3.3 `bool murasaki::DebuggerUart::ReceiveCompleteCallback ( void *const ptr ) [virtual]`

#### Parameters

<i>ptr</i>	Pointer to UART_HandleTypeDef struct.
------------	---------------------------------------

#### Returns

true: ptr matches with UART device and handle the call back. false : doesn't match.

A call back to notify the end of entire block transfer. This is considered as the end of DMA based receiving. The context have to be interrupt.

This member function checks whether the given ptr parameter matches its own device, and if matched, Release the waiting task and return true. If it doesn't match, just return false.

This method have to be called from [HAL\\_UART\\_RxCpltCallback\(\)](#). See STM32F7 HAL manual for detail

Implements [murasaki::UartStrategy](#).

### 7.14.3.4 `void murasaki::DebuggerUart::SetHardwareFlowControl ( UartHardwareFlowControl control ) [virtual]`

#### Parameters

<i>control</i>	The control mode.
----------------	-------------------

Before calling this method, all transmission and receive activities have to be finished. This is responsibility of the programmer.

Note this method is NOT re-entrant. In other word, this member function can be called from both task and interrupt context.

Reimplemented from [murasaki::UartStrategy](#).

### 7.14.3.5 `void murasaki::DebuggerUart::SetSpeed ( unsigned int baud_rate ) [virtual]`

#### Parameters

<i>baud_rate</i>	BAUD rate ( 110, 300,... 57600,... )
------------------	--------------------------------------

Before calling this method, all transmission and receive activities have to be finished. This is responsibility of the programmer.

Note this method is NOT re-entrant. In other word, this member function can be called from both task and interrupt context.

Reimplemented from [murasaki::UartStrategy](#).

### 7.14.3.6 `murasaki::UartStatus murasaki::DebuggerUart::Transmit ( const uint8_t * data, unsigned int size, unsigned int timeout_ms ) [virtual]`

## Parameters

<i>data</i>	Data buffer to be transmitted.
<i>size</i>	The count of the data ( byte ) to be transfered. Must be smaller than 65536
<i>timeout_ms</i>	Time out limit by milliseconds.

## Returns

Always returns OK

Transmit given data buffer through an UART device.

The transmission mode is synchronous. That means, function returns when all data has been transmitted, except timeout. Passing [murasaki::kwmsIndefinitely](#) to the parameter `timeout_ms` orders not to return until complete transmission. Other value of `timeout_ms` parameter specifies the time out by millisecond. If time out happen, function returns false. If not happen, it returns true.

This function is exclusive. Internally the function is guarded by mutex. Then this function is thread safe. This function is forbidden to call from ISR.

Implements [murasaki::UartStrategy](#).

**7.14.3.7** `bool murasaki::DebuggerUart::TransmitCompleteCallback ( void *const ptr )` [virtual]

## Parameters

<i>ptr</i>	Pointer to UART_HandleTypeDef struct.
------------	---------------------------------------

## Returns

true: ptr matches with UART device and handle the call back. false : doesn't match.

A call back to notify the end of entire block transfer. This is considered as the end of DMA based transmission. The context have to be interrupt.

This member function checks whether the given ptr parameter matches its own device, and if matched, Release the waiting task and return true. If it doesn't match, just return false.

This method have to be called from [HAL\\_UART\\_TxCpltCallback\(\)](#). See STM32F7 HAL manual for detail

Implements [murasaki::UartStrategy](#).

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

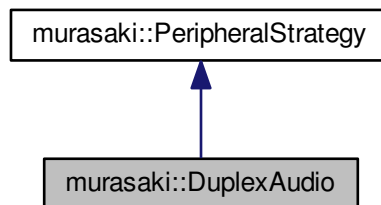
- [debuggeruart.hpp](#)
- [debuggeruart.cpp](#)

## 7.15 murasaki::DuplexAudio Class Reference

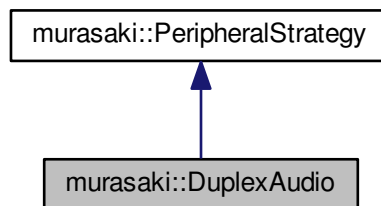
Stereo Audio is served by this class.

```
#include <duplexaudio.hpp>
```

Inheritance diagram for murasaki::DuplexAudio:



Collaboration diagram for murasaki::DuplexAudio:



### Public Member Functions

- [DuplexAudio](#) ([murasaki::AudioPortAdapterStrategy](#) \*peripheral\_adapter, unsigned int channel\_length)  
*Constructor.*
- virtual [~DuplexAudio](#) ()  
*Destructor.*
- void [TransmitAndReceive](#) (float \*tx\_left, float \*tx\_right, float \*rx\_left, float \*rx\_right)  
*Stereo audio transmission/receiving.*
- void [TransmitAndReceive](#) (float \*\*tx\_channels, float \*\*rx\_channels, unsigned int tx\_num\_of\_channels, unsigned int rx\_num\_of\_channels)  
*Multi channel audio transmission/receiving.*
- bool [DmaCallback](#) (void \*peripheral, unsigned int phase)  
*Callback function on the RX DMA interrupt.*
- virtual bool [HandleError](#) (void \*peripheral)  
*Call this function from the interrupt handler.*
- virtual bool [Match](#) (void \*peripheral\_handle)

### Protected Member Functions

- virtual void \* [GetPeripheralHandle](#) ()  
*Dummy member function.*

#### 7.15.1 Detailed Description

This class provides an interface to the audio data flow. Also, the internal buffer allocation, multi-phase buffering, and synchronization are provided. The features are :

- Support from mono to multi-ch audio
- 32bit floating-point data buffer as an interface with the application.
- Data range is [-1.0, 1.0) as an interface with the application.
- Blocking and synchronous API
- Internal DMA operation.

Note: This class assumes the Fs and the data size on I2S of the TX and RX are the same, and both Tx and RX are fully synchronized. Also, this class assumes that the data size on I2S is more significant than the 8bit.

Internally, this class provides a multi-buffers DMA operation between the audio peripheral and caller algorithm. The critical key API is the [TransmitAndReceive\(\)](#) member function. This function provides several key operations

- Multiple-buffer operation to allow a background DMA transfer during caller task is processing data.
- Data conversion and scaling between caller's floating-point data and DMA's integer data.
- Synchronization between [TransmitAndReceive\(\)](#) and DMA by [DmaCallback\(\)](#).

Thus, the user doesn't need to care about the above things.

Because of the complicated audio data structure, there are several terminologies that a programmer must know.

- Word : An atomic data of audio sample. For example, stereo sample has two word. Note that in [murasaki::DuplexAudio](#), the size of a word is given from [murasaki::AudioPortAdapterStrategy](#).
- Channel: Input / Output port of audio. For example, the stereo audio has two channels named left and right. The 5.1 surround audio has 6 channels.
- Phase: State of DMA. Usually, audio DMA is configured as double or triple buffered to avoid the gap of the sound. The index of the DMA buffer is called a phase. For example, the double buffer DMA can be phase 0 or 1 and incremented as modulo 2.

The number of phases is specified to the constructor, by the programmer. This phase has to be aligned with hardware.

#### 7.15.2 Constructor & Destructor Documentation

- ##### 7.15.2.1 `murasaki::DuplexAudio::DuplexAudio ( murasaki::AudioPortAdapterStrategy * peripheral_adapter, unsigned int channel_length )`

**Parameters**

<i>peripheral_adapter</i>	Pointer to the audio interface peripheral class
<i>channel_length</i>	Specify how many data are in one channel buffer.

Initialize the internal variables and allocate the buffer based on the given parameters.

The *channel\_length* parameter specifies the number of the data in one channel. Where channel is the independent audio data stream. For example, a stereo data has 2 channels named left and right.

**7.15.3 Member Function Documentation****7.15.3.1 bool murasaki::DuplexAudio::DmaCallback ( void \* *peripheral*, unsigned int *phase* )****Parameters**

<i>peripheral</i>	pointer to the peripheral device.
<i>phase</i>	0 or 1, ..., numPhase-1. The index of the buffer in the multi-buffer DMA.

**Returns**

True if the peripheral matches with own peripheral which was given by constructor. Otherwise false.

For each time RX DMA finish the transfer, interrupt should raised. This callback is designed to be called from that interrupt handler.

The interrupt must have phase. For example, for the double buffer DMA, it should have phase 0 and 1. For the triple buffer, it should have phase 0, 1, and 2. The maximum phase is defined by the *num\_dma\_phases* - 1, where *num\_dma\_phases* are given through the constructor parameter.

In some system, the interrupts have explicit phase information. For example, there are half-way-interrupt and end-of-buffer interrupt. In such the system, interrupt should give the phase parameter.

In certain system, the interrupts don't have explicit phase information. For example, only one interrupt happens on both half way and end of buffer. In this case, [AudioPortAdapterStrategy::DetectPhase](#) of the derived class must detect the phase. So, interrupt doesn't need to give the meaningful phase through this member function..

This function returns if peripheral parameter is match with the one passed by the constructor.

This member function have to be call from the HAL call backs of the SAI/I2S. In case of the SAI :

```
// Halfway
void HAL_SAI_RxHalfCpltCallback(SAI_HandleTypeDef * hsai) {
    if (murasaki::platform.audio->DmaCallback(hsai, 0)) // second parameter is 0 for the halfway
        return;
}

// Complete
void HAL_SAI_RxCpltCallback(SAI_HandleTypeDef * hsai) {
    if (murasaki::platform.audio->DmaCallback(hsai, 1)) // second parameter is 1 for the complete
        return;
}
```

And in case of I2S :

```
// Half way
void HAL_I2S_RxHalfCpltCallback(I2S_HandleTypeDef *hi2s) {
    if (murasaki::platform.audio->DmaCallback(hi2s, 0)) // second parameter is 0 for the halfway
        return;
}

// Complete
void HAL_I2S_RxCpltCallback(I2S_HandleTypeDef *hi2s) {
    if (murasaki::platform.audio->DmaCallback(hi2s, 1)) // second parameter is 1 for the complete
        return;
}
```

### 7.15.3.2 void \* murasaki::DuplexAudio::GetPeripheralHandle ( ) [protected],[virtual]

#### Returns

nothing

nullptr Do nothing. cause assertion fail.

Implements [murasaki::PeripheralStrategy](#).

### 7.15.3.3 bool murasaki::DuplexAudio::HandleError ( void \* *peripheral* ) [virtual]

#### Parameters

<i>peripheral</i>	pointer to the peripheral device.
-------------------	-----------------------------------

#### Returns

True if the peripheral matches with own peripheral which was given by constructor. Otherwise false.

This function calls the [AudioPortAdapterStrategy::HandleError\(\)](#) which knows how to handle. Usually, this error call back is unable to recover. So, assertion may be triggered.

This member function have to be called from the error call back of SAI/I2S HAL

```
void HAL_SAI_ErrorCallback(SAI_HandleTypeDef * hsai) {
    if (murasaki::platform.audio->HandleError(hsai))
        return;
}

void HAL_I2S_ErrorCallback(I2S_HandleTypeDef *hi2s) {
    if (murasaki::platform.audio->HandleError(hi2s))
        return;
}
```

### 7.15.3.4 bool murasaki::DuplexAudio::Match ( void \* *peripheral\_handle* ) [virtual]

Check if audio port peripheral handle matched with given handle.

#### Parameters

<i>peripheral_handle</i>	
--------------------------	--

**Returns**

true if match, false if not match.

Reimplemented from [murasaki::PeripheralStrategy](#).

**7.15.3.5** void [murasaki::DuplexAudio::TransmitAndReceive](#) ( float \* *tx\_left*, float \* *tx\_right*, float \* *rx\_left*, float \* *rx\_right* )

**Parameters**

<i>tx_left</i>	Pointer to the left channel TX buffer
<i>tx_right</i>	Pointer to the right channel TX buffer
<i>rx_left</i>	Pointer to the left channel RX buffer
<i>rx_right</i>	Pointer to the right channel RX buffer

Synchronous API. Inside this member function,

1. Wait for a complete of the RX data transfer by waiting for the [DmaCallback\(\)](#).
2. Given tx\_channels buffers are scaled and copied to the DMA buffer.
3. Scale the data in DMA buffer and copy to rx\_channels buffers.

And then returns.

Following is the typical usage of this function.

```
#define CH_LEN 48

float tx_left_ch_buf[CH_LEN];
float tx_right_ch_buf[CH_LEN];
float rx_left_ch_buf[CH_LEN];
float rx_right_ch_buf[CH_LEN];

while(1)
{
    // prepare TX data into tx_left_ch_buf and tx_right_ch_buf
    ...
    murasaki::platform.audio->TransmitAndReceive(
        tx_left_ch_buf,
        tx_right_ch_buf,
        rx_left_ch_buf,
        rx_right_ch_buf );

    // process RX data in rx_left_ch_buf and rx_right_ch_buf
    ...
}
```

**7.15.3.6** void [murasaki::DuplexAudio::TransmitAndReceive](#) ( float \*\* *tx\_channels*, float \*\* *rx\_channels*, unsigned int *tx\_num\_of\_channels*, unsigned int *rx\_num\_of\_channels* )

**Parameters**

<i>tx_channels</i>	Array of pointers. The number of the array element have to be same with the number of channel. Each pointer points the TX channel buffers.
<i>rx_channels</i>	Array of pointers. The number of the array element have to be same with the number of channel. Each pointer points the RX channel buffers.
<i>tx_num_of_channels</i>	Any number which is smaller than or equal to num_of_channels given audio peripheral adapter.
<i>rx_num_of_channels</i>	Any number which is smaller than or equal to num_of_channels given audio peripheral adapter.



Synchronous API. Inside this member function,

1. wait for the complete of the RX data transfer by waiting for the [DmaCallback\(\)](#).
2. Given tx\_channels buffers are scaled and copied to the DMA buffer.
3. Scale the data in DMA buffer and copy to rx\_channels buffers.

And then returns.

This function is a base for the another public TransmitAndRecieve().

```
#define NUM_CH 8
#define CH_LEN 48

float * tx_channels_array[NUM_CH];
float * rx_channels_array[NUM_CH];

tx_channles_array[0] = new float[CH_LEN];
tx_channles_array[1] = new float[CH_LEN];
tx_channles_array[2] = new float[CH_LEN];
...
tx_channles_array[NUM_CH-1] = new float[CH_LEN];

rx_channles_array[0] = new float[CH_LEN];
rx_channles_array[1] = new float[CH_LEN];
rx_channles_array[2] = new float[CH_LEN];
...
rx_channles_array[NUM_CH-1] = new float[CH_LEN];

while(1)
{
    // prepare TX data into rx_channlels_array.
    ...
    murasaki::platform.audio->TransmitAndReceive(
        tx_channels_array,
        rx_channels_array,
        NUM_CH,
        NUM_CH );

    // process RX data in rx_channles_array
    ...
}
```

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

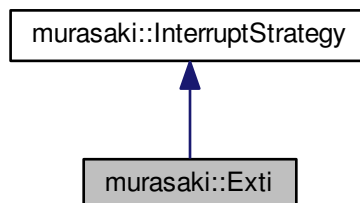
- [duplexaudio.hpp](#)
- [duplexaudio.cpp](#)

## 7.16 murasaki::Exti Class Reference

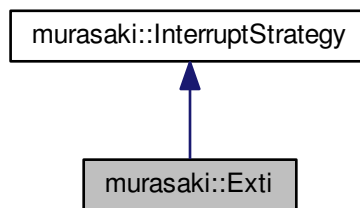
EXTI wrapper class.

```
#include <exti.hpp>
```

Inheritance diagram for murasaki::Exti:



Collaboration diagram for `murasaki::Exti`:



#### Public Member Functions

- `Exti` (unsigned int line)  
*Constructor.*
- virtual void `Enable` ()  
*Enable interrupt.*
- virtual void `Disable` ()  
*Disable interrupt.*
- virtual `murasaki::InterruptStatus Wait` (unsigned int timeout=`murasaki::kwmsIndefinitely`)
- virtual bool `Release` (unsigned int line)
- virtual bool `Match` (unsigned int line)

#### Static Public Member Functions

- static bool `isReady` ()  
*Show interruptible or not global.*

#### 7.16.1 Detailed Description

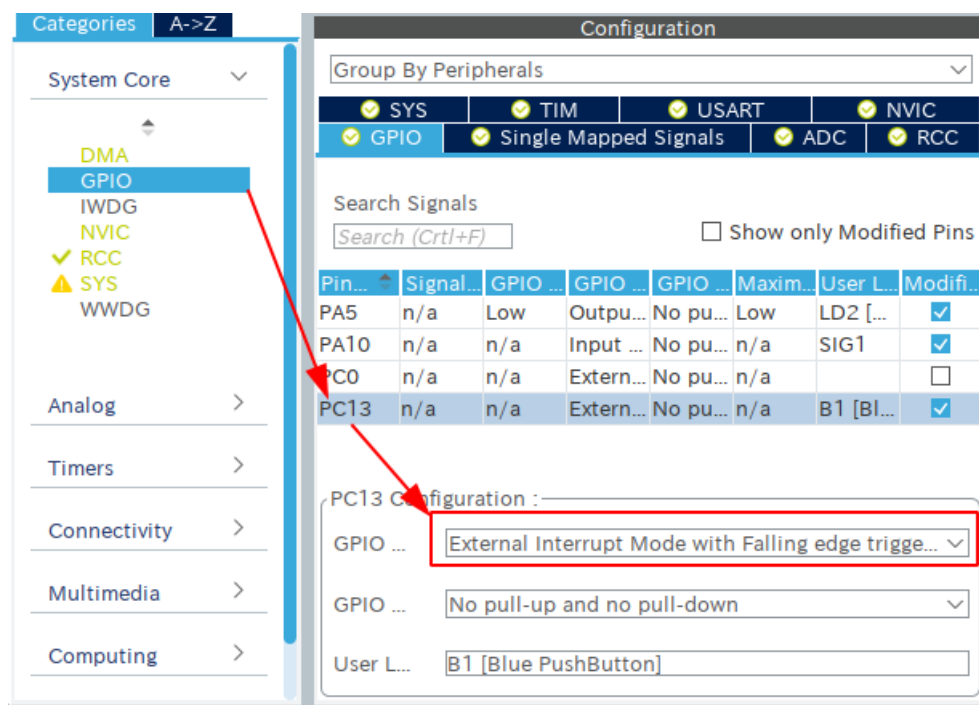
This class allows enabling/disabling EXTI interrupt from GPIO. Only EXTI\_0 to EXTI\_16 is supported.

In addition to the disabling /enabling interrupt, this class provide a simple synchronization between ISR and task, by Release and Wait function.

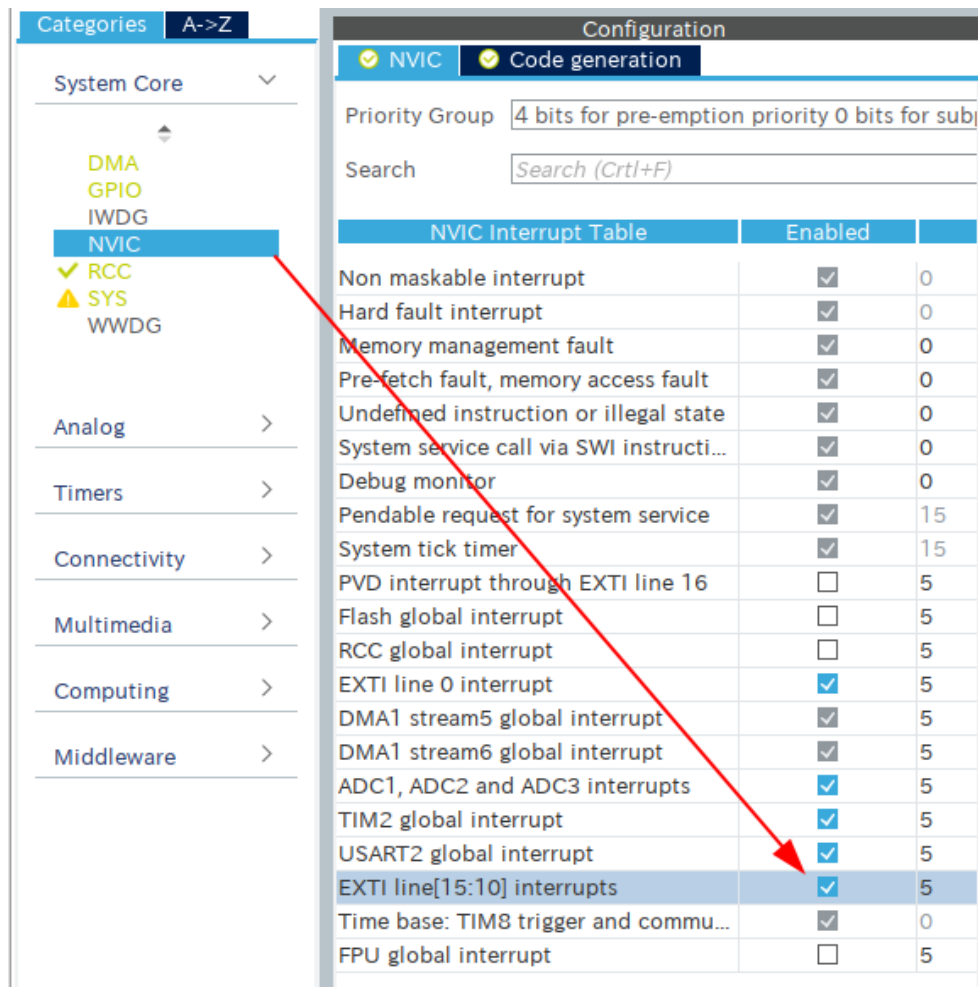
Because each instance of this class can handle only one line, to control the 16 input, you need to create 16 instances of this class.

## Configuration

To configure an EXTI, select the GPIO and line to use. And then, set it as external interrupt. In this example, we configure the EXTI13. You can give your name to this line by the User Label field. In this example, we name this EXTI13 as B1.



Next, you must enable the relevant NVIC interrupt line. In this example, we enable the NIVC of EXTI15:10 ( for EXTI 13 ).



### Creating a peripheral object

To create an EXTI object, create a class variable from `Exti` type. The parameter of the constructor should be a macro constant for the EXTI line, which is generated by CubeIDE.

```
my_exti = new murasaki::Exti(B1_pin);
```

### Waiting interrupt

To wait for an interrupt, use `Exti::Wait()` member function in a task.

```
my_exti.Wait();
```

### Interrupt handling

Interrupt is handled automatically. Programmer doesn't need to care.

## 7.16.2 Constructor & Destructor Documentation

### 7.16.2.1 murasaki::Exti::Exti ( unsigned int line )

## Parameters

<i>line</i>	EXTI line identifier. Use CubeMX/IDE created value (***_pin) or EXTI_LINE_**
-------------	--

The parameter passing should be like following. The B1\_Pin is the pin identifier of the EXTI generated by CubeIDE.

```
murasaki::platform.exti_b1 = new murasaki::Exti(B1_Pin);
```

Or, pre-defined EXTI line identifier can be passed.

```
murasaki::platform.exti_b1 = new murasaki::Exti(EXTI_LINE_13);
```

## 7.16.3 Member Function Documentation

## 7.16.3.1 bool murasaki::Exti::isReady ( ) [static]

## Returns

true if ready to handle the EXTI interrupt.

At the moment of 1Q/2020, CubeIDE generated code is interruptible from the first. Precisely speaking, the EXTI is set enable inside [main\(\)](#). That mean, the system is interruptible before the Murasaki EXTI callback is ready to accept the interrupt. ( Murasaki creates a list of EXTI object to accept the interrupt inside ExtiCallbackRepositorySingleton. If EXTI comes before the list is complete, assertion fails ).

7.16.3.2 bool murasaki::Exti::Match ( unsigned int *line* ) [virtual]

Check whether given line is handled by this object.

## Parameters

<i>line</i>	The identifier of the interrupt
-------------	---------------------------------

## Returns

true if line is matched with this EXTI. false if not matched.

Implements [murasaki::InterruptStrategy](#).

7.16.3.3 bool murasaki::Exti::Release ( unsigned int *line* ) [virtual]

Release the waiting task

## Parameters

<i>line</i>	Interrupt line bit map given from the <a href="#">HAL_GPIO_EXTI_Callback()</a>
-------------	--

**Returns**

true if line is matched with this EXTI. false if not matched.

Should be called from the interrupt call back.

The typical usage is to call from the [HAL\\_GPIO\\_EXTI\\_Callback\(\)](#) function which will be called from the HAL interrupt handler.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    // Check whether appropriate interrupt or not
    if (murasaki::platform.exti_b1->Release(GPIO_Pin))
        return;
}
```

Implements [murasaki::InterruptStrategy](#).

#### 7.16.3.4 **murasaki::InterruptStatus** **murasaki::Exti::Wait** ( unsigned int *timeout* = **murasaki::kwmsIndefinitely** ) [virtual]

Wait for an interrupt from task.

**Parameters**

<i>timeout</i>	time out parameter
----------------	--------------------

**Returns**

kisOK : Released by interrupt. kisTimeOut : released by timeout.

Implements [murasaki::InterruptStrategy](#).

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

- [exti.hpp](#)
- [exti.cpp](#)

## 7.17 **murasaki::ExtiCallbackRepositorySingleton** Class Reference

Repository of the EXTI handling objects.

```
#include <exticallbackrepositorysingleton.hpp>
```

**Public Member Functions**

- void [AddExtiObject](#) ([murasaki::InterruptStrategy](#) \*exti\_object)  
*Add the given object to the internal list of the EXTI.*
- [murasaki::InterruptStrategy](#) \* [GetExtiObject](#) (unsigned int line)  
*Search an EXTI responding object.*

**Static Public Member Functions**

- static [ExtiCallbackRepositorySingleton](#) \* [GetInstance](#) ()  
*Return a singleton instance.*

**7.17.1 Detailed Description**

A Singleton design repository. This is Murasaki internal only. Do not use in the application.

This class object stores the list of the EXTI callback responding objects. The registration is done in the client classe's construction phase. So, it is invisible from the application programmer.

For each EXTI, the callback routine calls the [GetExtiObject\(\)](#) with given peripheral handler. The [GetExtiHandler\(\)](#) routine search for the list whether responding object exists or not. If exists, return that object.

Only EXTI is supported by this class.

This class is not thread safe.

**7.17.2 Member Function Documentation****7.17.2.1 void murasaki::ExtiCallbackRepositorySingleton::AddExtiObject ( murasaki::InterruptStrategy \* *exti\_object* )****Parameters**

<i>exti_object</i>	Add this object to the list of the EXTI.
--------------------	--

Store the given object to the internal list. This member function is called from only inside [InterruptStrategy](#) or its sub classes.

**7.17.2.2 murasaki::InterruptStrategy \* murasaki::ExtiCallbackRepositorySingleton::GetExtiObject ( unsigned int *line* )****Parameters**

<i>line</i>	The line cause interrupt.
-------------	---------------------------

**Returns**

Matched object pointer.

If the object is not found, Assertion failed.

**7.17.2.3 ExtiCallbackRepositorySingleton \* murasaki::ExtiCallbackRepositorySingleton::GetInstance ( )**  
[static]**Returns**

Instance of this class. For each time of calling, this function returns identical pointer.

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

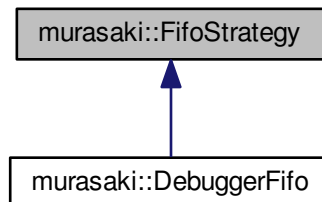
- [exticallbackrepositorysingleton.hpp](#)
- [exticallbackrepositorysingleton.cpp](#)

## 7.18 murasaki::FifoStrategy Class Reference

Basic FIFO without thread safe.

```
#include <fifostrategy.hpp>
```

Inheritance diagram for murasaki::FifoStrategy:



### Public Member Functions

- `FifoStrategy` (unsigned int buffer\_size)  
*Create an internal buffer.*
- virtual `~FifoStrategy` ()  
*Delete an internal buffer.*
- virtual unsigned int `Put` (uint8\_t const data[], unsigned int size)  
*Put the data into the internal buffer.*
- virtual unsigned int `Get` (uint8\_t data[], unsigned int size)  
*Get the data from the internal buffer.*

### Protected Attributes

- unsigned int const `size_of_buffer_`  
*Size of the internal buffer [byte].*
- uint8\_t \*const `buffer_`  
*The internal buffer. Allocated by constructor.*

#### 7.18.1 Detailed Description

Fundamental FIFO. No blocking, not thread-safe.

The `Put()` member function returns with "copied" data count. If the internal buffer is full, it returns without copy data.

The `Get()` member function returns with "copied" data count and data. If the internal buffer is empty, it returns without copy data.

#### 7.18.2 Constructor & Destructor Documentation

##### 7.18.2.1 murasaki::FifoStrategy::FifoStrategy ( unsigned int buffer\_size )



## Parameters

<i>buffer_size</i>	Size of the internal buffer to be allocated [byte]
--------------------	--

Allocate the internal buffer with given *buffer\_size*. The contents is not initialized.

## 7.18.3 Member Function Documentation

7.18.3.1 unsigned int murasaki::FifoStrategy::Get ( uint8\_t *data*[], unsigned int *size* ) [virtual]

## Parameters

<i>data</i>	Data buffer to receive from the internal buffer
<i>size</i>	Size of the data parameter.

## Returns

The count of copied data. 0, if the internal buffer is empty

Reimplemented in [murasaki::DebuggerFifo](#).

7.18.3.2 unsigned int murasaki::FifoStrategy::Put ( uint8\_t const *data*[], unsigned int *size* ) [virtual]

## Parameters

<i>data</i>	Data to be copied to the internal buffer
<i>size</i>	Data count to be copied

## Returns

The count of copied data. 0, if the internal buffer is full.

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

- [fifostrategy.hpp](#)
- [fifostrategy.cpp](#)

## 7.19 murasaki::GPIO\_type Struct Reference

A structure to en-group the GPIO port and GPIO pin.

```
#include <bitout.hpp>
```

## Public Attributes

- GPIO\_TypeDef \* [port\\_](#)  
*The port.*
- uint16\_t [pin\\_](#)  
*The number of pin.*

### 7.19.1 Detailed Description

This struct is used in the [BitIn](#) class and [BitOut](#) class. These classes returns a pointer to the variable of this type, as return value of the `GetPeripheralHandle()` member function.

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

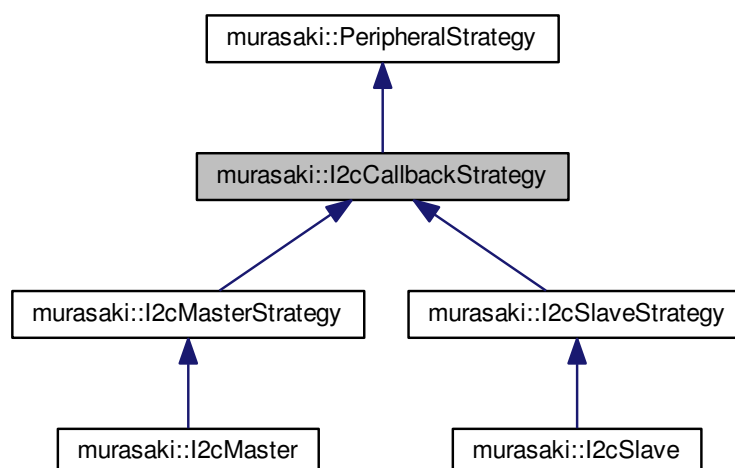
- [bitout.hpp](#)

## 7.20 murasaki::I2cCallbackStrategy Class Reference

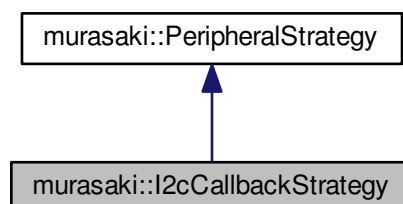
Definition of the root class of I2C .

```
#include <i2ccallbackstrategy.hpp>
```

Inheritance diagram for `murasaki::I2cCallbackStrategy`:



Collaboration diagram for `murasaki::I2cCallbackStrategy`:



## Public Member Functions

- virtual bool [TransmitCompleteCallback](#) (void \*ptr)=0  
*Call back to be called notify the transfer is complete.*
- virtual bool [ReceiveCompleteCallback](#) (void \*ptr)=0  
*Call back to be called for entire block transfer is complete.*
- virtual bool [HandleError](#) (void \*ptr)=0  
*Handling error report of device.*

## Additional Inherited Members

## 7.20.1 Detailed Description

This class specify the unified callbacks between the master and slave devices.

## 7.20.2 Member Function Documentation

7.20.2.1 virtual bool murasaki::I2cCallbackStrategy::HandleError ( void \* *ptr* ) [pure virtual]

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a device control
------------	---

## Returns

true if ptr matches with device and handle the error. false if ptr doesn't match This member function is a callback to detect error.

The error handling is depend on the implementation.

Implemented in [murasaki::I2cMaster](#), and [murasaki::I2cSlave](#).

7.20.2.2 virtual bool murasaki::I2cCallbackStrategy::ReceiveCompleteCallback ( void \* *ptr* ) [pure virtual]

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a peripheral control
------------	---

## Returns

true: ptr matches with peripheral and handle the call back. false : doesn't match.

This member function is a callback to notify the end of the entire block or byte transfer. The definition of calling timing depends on the implementation. You must call this function from an ISR.

Typically, an implementation may check whether the given ptr parameter matches its device, and if matched, handle it and return true. If it doesn't match, return false.

Implemented in [murasaki::I2cMaster](#), and [murasaki::I2cSlave](#).

7.20.2.3 virtual bool murasaki::I2cCallbackStrategy::TransmitCompleteCallback ( void \* *ptr* ) [pure virtual]

**Parameters**

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a peripheral control
------------	---

**Returns**

true: ptr matches with peripheral and handle the call back. false : doesn't match.

A call back to notify the end of entire block or byte transfer. The definition of calling timing is depend on the implementation. This is called from an DMA ISR.

Typically, an implementation may check whether the given ptr parameter matches its own device, and if matched, handle it and return true. If it doesn't match, just return false.

Implemented in [murasaki::I2cMaster](#), and [murasaki::I2cSlave](#).

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

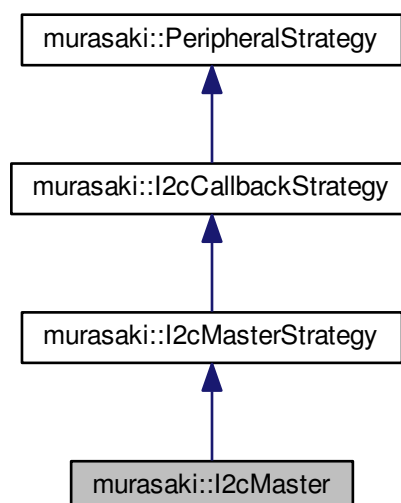
- [i2ccallbackstrategy.hpp](#)

**7.21 murasaki::I2cMaster Class Reference**

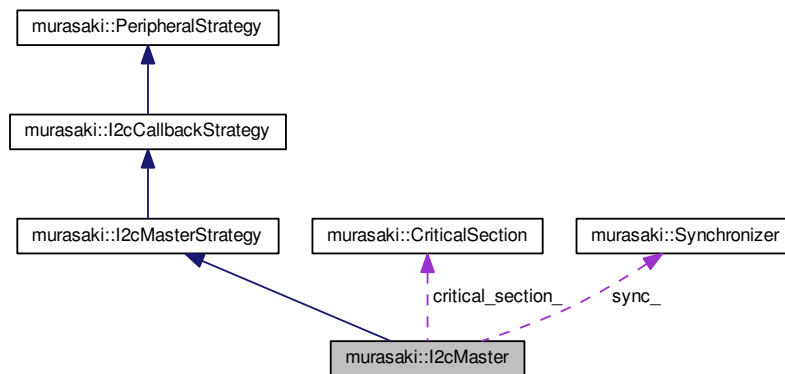
Thread safe, synchronous, and blocking IO. Encapsulating I2C master. Based on STM32Cube HAL driver and FreeRTOS.

```
#include <i2cmaster.hpp>
```

Inheritance diagram for murasaki::I2cMaster:



Collaboration diagram for murasaki::I2cMaster:



### Public Member Functions

- [I2cMaster](#) (I2C\_HandleTypeDef \*i2c\_handle)  
*Constructor.*
- virtual [~I2cMaster](#) ()  
*Destructor.*
- virtual [murasaki::I2cStatus Transmit](#) (unsigned int addr, const uint8\_t \*tx\_data, unsigned int tx\_size, unsigned int \*transferred\_count, unsigned int timeout\_ms)  
*Thread safe, synchronous transmission over I2C.*
- virtual [murasaki::I2cStatus Receive](#) (unsigned int addr, uint8\_t \*rx\_data, unsigned int rx\_size, unsigned int \*transferred\_count, unsigned int timeout\_ms)  
*Thread safe, synchronous receiving over I2C.*
- virtual [murasaki::I2cStatus TransmitThenReceive](#) (unsigned int addr, const uint8\_t \*tx\_data, unsigned int tx\_size, uint8\_t \*rx\_data, unsigned int rx\_size, unsigned int \*tx\_transferred\_count, unsigned int \*rx\_transferred\_count, unsigned int timeout\_ms)  
*Thread safe, synchronous transmission and then receiving over I2C.*
- virtual bool [TransmitCompleteCallback](#) (void \*ptr)  
*Call back to be called notify the transfer is complete.*
- virtual bool [ReceiveCompleteCallback](#) (void \*ptr)  
*Call back to be called for entire block transfer is complete.*
- virtual bool [HandleError](#) (void \*ptr)  
*Error handling.*

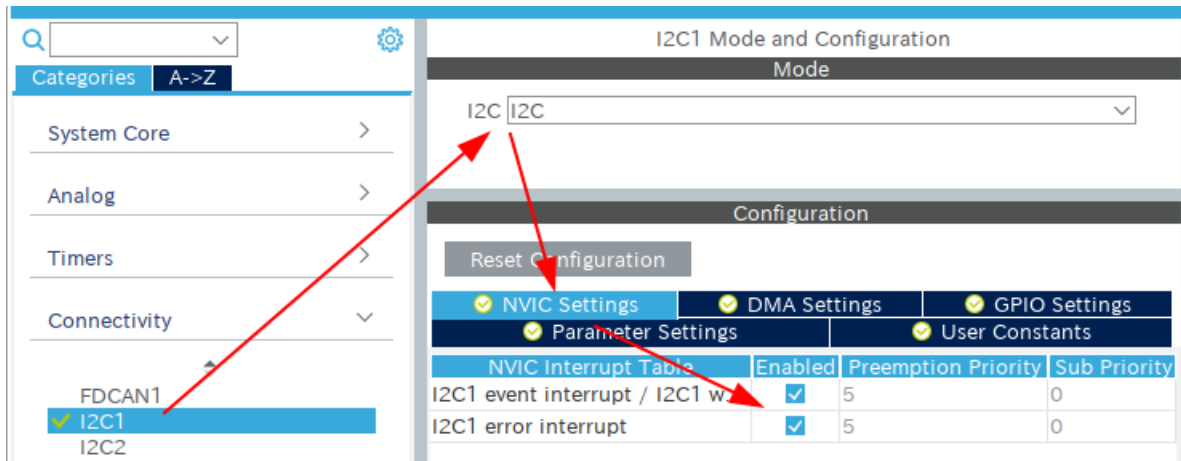
### Additional Inherited Members

#### 7.21.1 Detailed Description

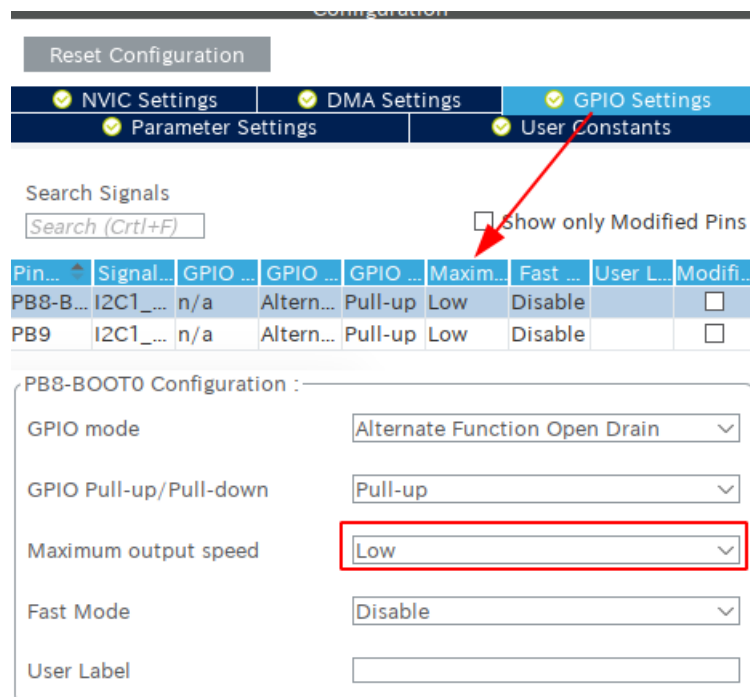
The [I2cMaster](#) class is the wrapper of the I2C controller.

## Configuration

To configure the I2C peripheral as master, chose I2C peripheral in the Device Configuration Tool of the CubeIDE. Set it as I2C mode, and enable NVIC interrupt.



Also, pay attention to the I2C Maximum Output Speed. The default setting by CubeIDE may not be appropriate to your circuit. You should check with an oscilloscope.



## Creating a peripheral object

To use the `I2cMaster` class, create an instance with `I2C_HandleTypeDef *` type pointer. For example, to create an instance for the I2C3 peripheral :

```
my_i2c3 = new murasaki::I2cMaster(&hi2c3);
```

Where `hi2c3` is the handle generated by CubeIDE for I2C3 peripheral.

### Handling an interrupt

Interrupt is handled automatically. Programmer doesn't need to care.

### Transmitting and Receiving

Once the instance and callback are correctly prepared, we can use the Tx/Rx member function.

The `I2cMaster::Transmit()` member function is a synchronous function. A programmer can specify the timeout by the `timeout_ms` parameter. By default, this parameter is set by `murasaki::kwmsIndefinitely` which specifies eternal wait.

The `I2cMaster::Receive()` member function is a synchronous function. A programmer can specify the timeout by `timeout_ms` parameter. By default, this parameter is set by `murasaki::kwmsIndefinitely` which specifies eternal wait.

The `I2cMaster::TransmitThenReceive()` member function is synchronous function. A programmer can specify the timeout by the `timeout_ms` parameter. By default, this parameter is set by `kwmsIndefinitely` which species never time out.

You can call these 3 member functions from only the task context. If you l them in the ISR context, the result is unknown.

Note: In case a time out occurs during transmit / receive, this implementation calls `HAL_I2C_MASTER_ABORT_IT()`. But it is unknown whether this is the right thing to do. The HAL reference of the STM32F7 is not clear for this case. For example, it doesn't tell what a programmer has to do to stop the transfer in the middle. And also, it doesn't tell what's happen if a programmer call `HAL_I2C_MASTER_ABORT_IT()`.

According to the source code of the `HAL_I2C_MASTER_ABORT_IT()`, no interrupt will be raised by this API call.

## 7.21.2 Constructor & Destructor Documentation

### 7.21.2.1 murasaki::I2cMaster::I2cMaster ( I2C\_HandleTypeDef \* i2c\_handle )

#### Parameters

<i>i2c_handle</i>	Peripheral handle created by CubeMx
-------------------	-------------------------------------

## 7.21.3 Member Function Documentation

### 7.21.3.1 bool murasaki::I2cMaster::HandleError ( void \* ptr ) [virtual]

#### Parameters

<i>ptr</i>	Pointer to I2C_HandleTypeDef struct.
------------	--------------------------------------

#### Returns

true: ptr matches with device and handle the error. false : doesn't match.

This member function handles the error case based on the internal error code.

You must call this member function from `HAL_I2C_ErrorCallback()`.

```

void HAL_I2C_ErrorCallback(I2C_HandleTypeDef * hi2c) {
    if (murasaki::platform.i2c_master->HandleError(hi2c))
        return;
    if (murasaki::platform.i2c_slave->HandleError(hi2c))
        return;
}

```

Implements [murasaki::I2cCallbackStrategy](#).

**7.21.3.2** `murasaki::I2cStatus murasaki::I2cMaster::Receive ( unsigned int addr, uint8_t * rx_data, unsigned int rx_size, unsigned int * transferred_count, unsigned int timeout_ms )` [virtual]

#### Parameters

<i>addr</i>	7bit address of the I2C device.
<i>rx_data</i>	Data array to transmit.
<i>rx_size</i>	Data counts[bytes] to transmit. Must be smaller than 65536
<i>transferred_count</i>	( Currently, Just ignored) the count of the bytes transferred during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

#### Returns

Result of the processing

This member function is programmed to run in the task context of RTOS. And also exclusive between the racing tasks. In other words, this function is thread-safe.

Followings are the return codes :

- [murasaki::ki2csOK](#) : All Receive completed.
- [murasaki::ki2csNak](#) : Receive terminated by NAK receiving.
- [murasaki::ki2csArbitrationLost](#) : Receive terminated by an arbitration error of the multi-master.
- [murasaki::ki2csBussError](#) : Receive terminated by bus error
- [murasaki::ki2csTimeOut](#) : Receive abort by timeout.
- other value : Unhandled error. I2C device are re-initialized.

Implements [murasaki::I2cMasterStrategy](#).

**7.21.3.3** `bool murasaki::I2cMaster::ReceiveCompleteCallback ( void * ptr )` [virtual]

#### Parameters

<i>ptr</i>	Pointer to I2C_HandleTypeDef struct.
------------	--------------------------------------

#### Returns

true: ptr matches with peripheral and handle the call back. false : doesn't match.

This callback function notifies the end of the entire block or byte transfer. You must call this function from ISR.



This function checks whether the given ptr parameter matches its device, and if matched, handle it and return true. If it doesn't match, return false. You must call this function from [HAL\\_I2C\\_MasterRxCpltCallback\(\)](#)

```
void HAL_I2C_MasterRxCpltCallback(I2C_HandleTypeDef * hi2c) {
    if (murasaki::platform.i2c_master->ReceiveCompleteCallback(hi2c))
        return;
}
```

Implements [murasaki::I2cCallbackStrategy](#).

**7.21.3.4 murasaki::I2cStatus** `murasaki::I2cMaster::Transmit ( unsigned int addr, const uint8_t * tx_data, unsigned int tx_size, unsigned int * transferred_count, unsigned int timeout_ms )` [virtual]

#### Parameters

<i>addr</i>	7bit address of the I2C device.
<i>tx_data</i>	Data array to transmit.
<i>tx_size</i>	Data counts[bytes] to transmit. Must be smaller than 65536
<i>transferred_count</i>	( Currently, Just ignored) the count of the bytes transferred during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

#### Returns

Result of the processing

This member function is programmed to run in the task context of RTOS. And also exclusive between the racing tasks. In other words, this function is thread-safe.

Followings are the return codes :

- [murasaki::ki2csOK](#) : All transmission completed.
- [murasaki::ki2csNak](#) : Transmission terminated by NAK receiving.
- [murasaki::ki2csArbitrationLost](#) : Transmission terminated by an arbitration error of the multi-master.
- [murasaki::ki2csBussError](#) : Transmission terminated by bus error
- [murasaki::ki2csTimeOut](#) : Transmission abort by timeout.
- other value : Unhandled error. I2C device are re-initialized.

Implements [murasaki::I2cMasterStrategy](#).

**7.21.3.5** `bool murasaki::I2cMaster::TransmitCompleteCallback ( void * ptr )` [virtual]

#### Parameters

<i>ptr</i>	Pointer to I2C_HandleTypeDef struct.
------------	--------------------------------------

**Returns**

true: ptr matches with peripheral and handle the call back. false : doesn't match.

This callback function notifies the end of the entire block or byte transfer. You must call this function from ISR.

This function checks whether the given ptr parameter matches its device, and if matched, handle it and return true. If it doesn't match, return false. You must call this function from [HAL\\_I2C\\_MasterTxCallback\(\)](#),

```
void HAL_I2C_MasterTxCallback(I2C_HandleTypeDef * hi2c)
{
    if (murasaki::platform.i2c_master->TransmitCompleteCallback(hi2c))
        return;
}
```

Implements [murasaki::I2cCallbackStrategy](#).

**7.21.3.6** [murasaki::I2cStatus](#) [murasaki::I2cMaster::TransmitThenReceive](#) ( unsigned int *addr*, const uint8\_t \* *tx\_data*, unsigned int *tx\_size*, uint8\_t \* *rx\_data*, unsigned int *rx\_size*, unsigned int \* *tx\_transferred\_count*, unsigned int \* *rx\_transferred\_count*, unsigned int *timeout\_ms* ) [virtual]

**Parameters**

<i>addr</i>	7bit address of the I2C device.
<i>tx_data</i>	Data array to transmit.
<i>tx_size</i>	Data counts[bytes] to transmit. Must be smaller than 65536
<i>rx_data</i>	Data array to transmit.
<i>rx_size</i>	Data counts[bytes] to transmit. Must be smaller than 65536
<i>tx_transferred_count</i>	( Currently, Just ignored) the count of the bytes transmitted during the API execution.
<i>rx_transferred_count</i>	( Currently, Just ignored) the count of the bytes received during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

**Returns**

Result of the processing

First, this member function to transmit the data, and then, by repeated start function, it receives data. The transmission device address and receiving device address is the same.

This member function is programmed to run in the task context of RTOS. And also exclusive between the racing tasks. In other words, this function is thread-safe.

Followings are the return codes :

- [murasaki::ki2csOK](#) : All transmission and receive completed.
- [murasaki::ki2csNak](#) : Transmission or receive terminated by NAK receiving.
- [murasaki::ki2csArbitrationLost](#) : Transmission or receive terminated by an arbitration error of the multi-master.
- [murasaki::ki2csBussError](#) : Transmission or receive terminated by bus error
- [murasaki::ki2csTimeOut](#) : Transmission or receive abort by timeout.
- other value : Unhandled error. I2C device are re-initialized.

Implements [murasaki::I2cMasterStrategy](#).

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

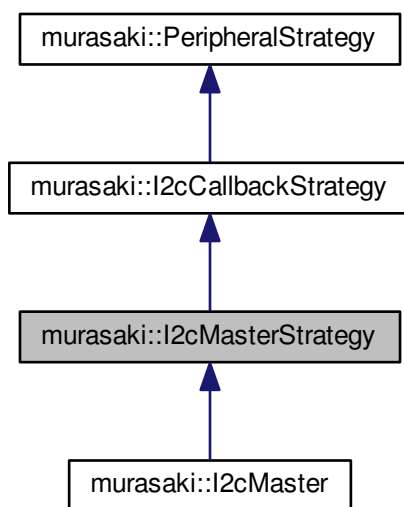
- [i2cmaster.hpp](#)
- [i2cmaster.cpp](#)

## 7.22 murasaki::I2cMasterStrategy Class Reference

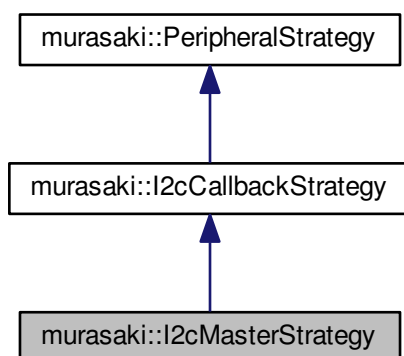
Definition of the root class of I2C master.

```
#include <i2cmasterstrategy.hpp>
```

Inheritance diagram for murasaki::I2cMasterStrategy:



Collaboration diagram for murasaki::I2cMasterStrategy:



## Public Member Functions

- virtual [murasaki::I2cStatus Transmit](#) (unsigned int *addr*, const uint8\_t \**tx\_data*, unsigned int *tx\_size*, unsigned int \**transferred\_count*=nullptr, unsigned int *timeout\_ms*=[murasaki::kwmsIndefinitely](#))=0  
*Thread safe, synchronous transmission over I2C.*
- virtual [murasaki::I2cStatus Receive](#) (unsigned int *addr*, uint8\_t \**rx\_data*, unsigned int *rx\_size*, unsigned int \**transferred\_count*=nullptr, unsigned int *timeout\_ms*=[murasaki::kwmsIndefinitely](#))=0  
*Thread safe, synchronous receiving over I2C.*
- virtual [murasaki::I2cStatus TransmitThenReceive](#) (unsigned int *addr*, const uint8\_t \**tx\_data*, unsigned int *tx\_size*, uint8\_t \**rx\_data*, unsigned int *rx\_size*, unsigned int \**tx\_transferred\_count*=nullptr, unsigned int \**rx\_transferred\_count*=nullptr, unsigned int *timeout\_ms*=[murasaki::kwmsIndefinitely](#))=0  
*Thread safe, synchronous transmission and then receiving over I2C.*

## Additional Inherited Members

### 7.22.1 Detailed Description

This member function is a prototype of the I2C master peripheral.

This prototype assumes the derived class will transmit/receive data in the task context on RTOS. And these member functions should be synchronous. That means, both member functions don't return until the transmit/receive terminates.

These two callback member functions synchronize with the interrupt, which tells the end of Transmit/Receive.

### 7.22.2 Member Function Documentation

- 7.22.2.1 `virtual murasaki::I2cStatus murasaki::I2cMasterStrategy::Receive ( unsigned int addr, uint8_t *rx_data, unsigned int rx_size, unsigned int * transferred_count = nullptr, unsigned int timeout_ms = murasaki::kwmsIndefinitely ) [pure virtual]`

#### Parameters

<i>addr</i>	7bit address of the I2C device.
<i>rx_data</i>	Data array to transmit.
<i>rx_size</i>	Data counts[bytes] to transmit.
<i>transferred_count</i>	the count of the bytes transferred during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

#### Returns

Result of the processing

This member function is programmed to run in the task context of RTOS. This function should be internally exclusive between multiple task access. In other words, it should be thread-safe.

Implemented in [murasaki::I2cMaster](#).

- 7.22.2.2 `virtual murasaki::I2cStatus murasaki::I2cMasterStrategy::Transmit ( unsigned int addr, const uint8_t *tx_data, unsigned int tx_size, unsigned int * transferred_count = nullptr, unsigned int timeout_ms = murasaki::kwmsIndefinitely ) [pure virtual]`

## Parameters

<i>addr</i>	7bit address of the I2C device.
<i>tx_data</i>	Data array to transmit.
<i>tx_size</i>	Data counts[bytes] to transmit.
<i>transferred_count</i>	the count of the bytes transferred during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

## Returns

Result of the processing

This member function is programmed to run in the task context of RTOS. This function should be internally exclusive between multiple task access. In other words, it should be thread-safe.

Implemented in [murasaki::I2cMaster](#).

```
7.22.2.3 virtual murasaki::I2cStatus murasaki::I2cMasterStrategy::TransmitThenReceive ( unsigned int addr,
const uint8_t * tx_data, unsigned int tx_size, uint8_t * rx_data, unsigned int rx_size, unsigned int *
tx_transferred_count = nullptr, unsigned int * rx_transferred_count = nullptr, unsigned int timeout_ms =
murasaki::kwmmsIndefinitely ) [pure virtual]
```

## Parameters

<i>addr</i>	7bit address of the I2C device.
<i>tx_data</i>	Data array to transmit.
<i>tx_size</i>	Data counts[bytes] to transmit.
<i>rx_data</i>	Data array to transmit.
<i>rx_size</i>	Data counts[bytes] to transmit.
<i>tx_transferred_count</i>	the count of the bytes transmitted during the API execution.
<i>rx_transferred_count</i>	the count of the bytes received during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

## Returns

Result of the processing

First, this member function to transmit the data, and then, by repeated start function, it receives data. The transmission device address and receiving device address is the same.

This member function is programmed to run in the task context of RTOS. And also exclusive between the racing tasks. In other words, this function is thread-safe.

Implemented in [murasaki::I2cMaster](#).

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

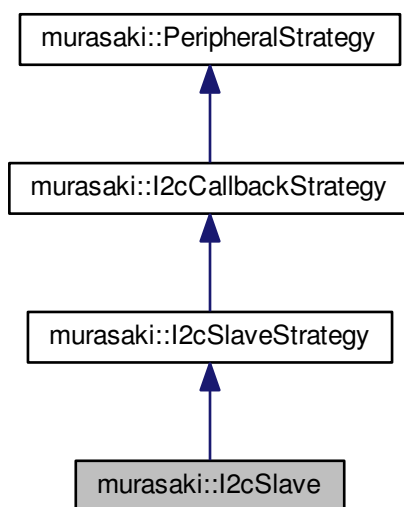
- [i2cmasterstrategy.hpp](#)

## 7.23 murasaki::I2cSlave Class Reference

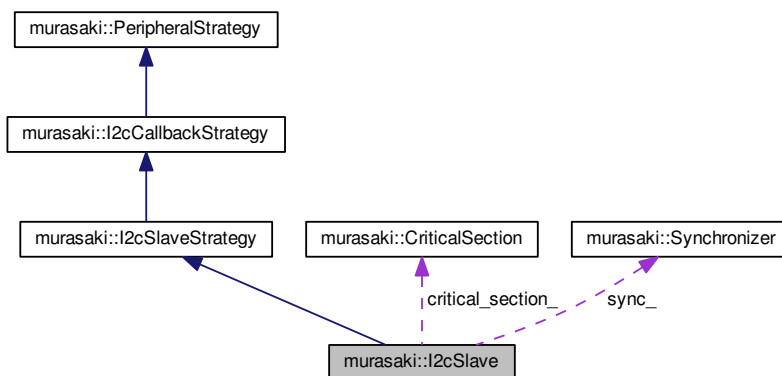
Thread safe, synchronous and blocking IO. Encapsulating I2C slave. Based on STM32Cube HAL driver and FreeRTOS.

```
#include <i2cslave.hpp>
```

Inheritance diagram for murasaki::I2cSlave:



Collaboration diagram for murasaki::I2cSlave:



## Public Member Functions

- virtual [murasaki::I2cStatus Transmit](#) (const uint8\_t \*tx\_data, unsigned int tx\_size, unsigned int \*transferred\_count, unsigned int timeout\_ms)  
*Thread safe, synchronous transmission over I2C.*
- virtual [murasaki::I2cStatus Receive](#) (uint8\_t \*rx\_data, unsigned int rx\_size, unsigned int \*transferred\_count, unsigned int timeout\_ms)  
*Thread safe, synchronous receiving over I2C.*
- virtual bool [TransmitCompleteCallback](#) (void \*ptr)  
*Call back to be called notify the transfer is complete.*
- virtual bool [ReceiveCompleteCallback](#) (void \*ptr)  
*Call back to be called for entire block transfer is complete.*
- virtual bool [HandleError](#) (void \*ptr)  
*Error handling.*

## Additional Inherited Members

## 7.23.1 Detailed Description

The [I2cSlave](#) class is the wrapper of the I2C controller.

## Configuration

The configuration is the same as the master. See [murasaki::I2cMaster](#) class.

## Creating a peripheral object

To use the [I2cSlave](#) class, create an instance with I2C\_HandleTypeDef \* type pointer. For example, to create an instance for the I2C3 peripheral :

```
my_i2c3 = new murasaki::I2cSlave(&hi2c3);
```

Where hi2c3 is the handle generated by CubeIDE for I2C3 peripheral.

## Handling an interrupt

Interrupt is handled automatically. Programmer doesn't need to care.

## Transmission and Receiving

Once the instance and callbacks are correctly prepared, we can use the Tx/Rx member function.

The [I2cSlave::Transmit\(\)](#) member function is a synchronous function. A programmer can specify the timeout by timeout\_ms parameter. By default, this parameter is set by kwmsIndefinitely which specifies eternal wait.

The [I2cSlave::Receive\(\)](#) member function is a synchronous function. A programmer can specify the timeout by timeout\_ms parameter. By default, this parameter is set by kwmsIndefinitely which specifies eternal wait. Both methods can be called from only the task context. If these are called in the ISR context, the result is unknown.

Note : In case an time out occurs during transmit / receive, this implementation calls HAL\_I2C\_DeInit()/HAL\_I2C\_Init(). But it is unknown whether this is the right thing to do. The HAL reference of the STM32F7 is not clear for this case. For example, it doesn't tell what programmer do to stop the transfer at the middle.

## 7.23.2 Member Function Documentation

## 7.23.2.1 bool murasaki::I2cSlave::HandleError ( void \* ptr ) [virtual]

**Parameters**

<i>ptr</i>	Pointer to I2C_HandleTypeDef struct.
------------	--------------------------------------

**Returns**

true: ptr matches with device and handle the error. false : doesn't match.

A handle to print out the error message.

Checks whether handle has error and if there is, print appropriate error. Then return.

This member function have to be called from [HAL\\_I2C\\_ErrorCallback\(\)](#)

```
void HAL_I2C_ErrorCallback(I2C_HandleTypeDef * hi2c) {
    if (murasaki::platform.i2c_master->HandleError(hi2c))
        return;
    if (murasaki::platform.i2c_slave->HandleError(hi2c))
        return;
}
```

Implements [murasaki::I2cCallbackStrategy](#).

**7.23.2.2** `murasaki::I2cStatus murasaki::I2cSlave::Receive ( uint8_t * rx_data, unsigned int rx_size, unsigned int * transfered_count, unsigned int timeout_ms )` [virtual]

**Parameters**

<i>rx_data</i>	Data array to transmit.
<i>rx_size</i>	Data counts[bytes] to transmit. Must be smaller than 65536
<i>transfered_count</i>	( Currently, Just ignored) the count of the bytes transferred during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

**Returns**

Result of the processing

This member function is programmed to run in the task context of RTOS. This should be internally exclusive between multiple task access. In other word, it should be thread save.

Following are the return code :

- [murasaki::ki2csOK](#) : All Receive completed.
- [murasaki::ki2csNak](#) : Receive terminated by NAK receiving.
- [murasaki::ki2csArbitrationLost](#) : Receive terminated by an arbitration error of the multi-master.
- [murasaki::ki2csBussError](#) : Receive terminated by bus error
- [murasaki::ki2csTimeOut](#) : Receive abort by timeout.
- other value : Unhandled error. I2C device are re-initialized.

Implements [murasaki::I2cSlaveStrategy](#).

**7.23.2.3** `bool murasaki::I2cSlave::ReceiveCompleteCallback ( void * ptr )` [virtual]



## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a peripheral control
------------	---

## Returns

true: ptr matches with peripheral and handle the call back. false : doesn't match.

A call back to notify the end of entire block or byte transfer. The definition of calling timing is depend on the implementation. This is called from an DMA ISR.

Typically, an implementation may check whether the given ptr parameter matches its own device, and if matched, handle it and return true. If it doesn't match, just return false.

This function have to be called from [HAL\\_I2C\\_SlaveRxCpltCallback\(\)](#)

```
void HAL_I2C_SlaveRxCpltCallback(I2C_HandleTypeDef * hi2c) {
    if (murasaki::platform.i2c_slave->ReceiveCompleteCallback(hi2c))
        return;
}
```

Implements [murasaki::I2cCallbackStrategy](#).

**7.23.2.4** `murasaki::I2cStatus murasaki::I2cSlave::Transmit ( const uint8_t * tx_data, unsigned int tx_size, unsigned int * transfered_count, unsigned int timeout_ms ) [virtual]`

## Parameters

<i>tx_data</i>	Data array to transmit.
<i>tx_size</i>	Data counts[bytes] to transmit. Must be smaller than 65536
<i>transfered_count</i>	( Currently, Just ignored) the count of the bytes transferred during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

## Returns

Result of the processing

This member function is programmed to run in the task context of RTOS. This should be internally exclusive between multiple task access. In other word, it should be thread save.

Following are the return code :

- [murasaki::ki2csOK](#) : All transmission completed.
- [murasaki::ki2csNak](#) : Transmission terminated by NAK receiving.
- [murasaki::ki2csArbitrationLost](#) : Transmission terminated by an arbitration error of the multi-master.
- [murasaki::ki2csBussError](#) : Transmission terminated by bus error
- [murasaki::ki2csTimeOut](#) : Transmission abort by timeout.
- other value : Unhandled error. I2C device are re-initialized.

Implements [murasaki::I2cSlaveStrategy](#).

**7.23.2.5** `bool murasaki::I2cSlave::TransmitCompleteCallback ( void * ptr ) [virtual]`

**Parameters**

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a peripheral control
------------	---

**Returns**

true: ptr matches with peripheral and handle the call back. false : doesn't match.

A call back to notify the end of entire block or byte transfer. The definition of calling timing is depend on the implementation. This is called from an DMA ISR.

Typically, an implementation may check whether the given ptr parameter matches its own device, and if matched, handle it and return true. If it doesn't match, just return false.

This member function have to be called from [HAL\\_I2C\\_SlaveTxCpltCallback\(\)](#)

```
void HAL_I2C_SlaveTxCpltCallback(I2C_HandleTypeDef * hi2c)
{
    if (murasaki::platform.i2c_slave->TransmitCompleteCallback(hi2c))
        return;
}
```

Implements [murasaki::I2cCallbackStrategy](#).

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

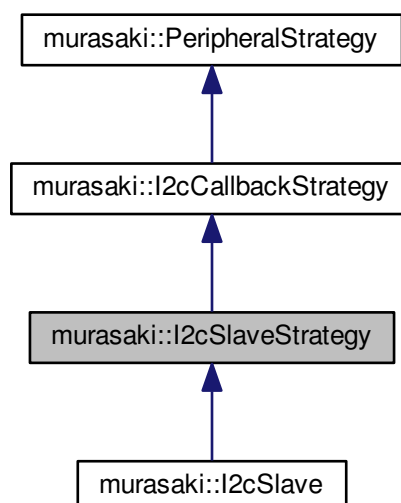
- [i2cslave.hpp](#)
- [i2cslave.cpp](#)

**7.24 murasaki::I2cSlaveStrategy Class Reference**

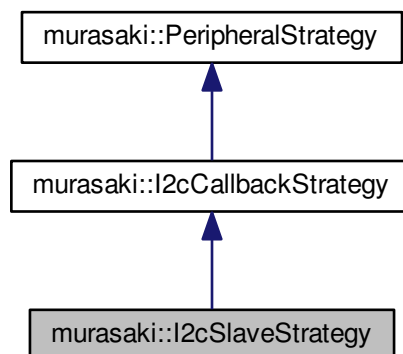
Definition of the root class of I2C Slave.

```
#include <i2cslavestrategy.hpp>
```

Inheritance diagram for murasaki::I2cSlaveStrategy:



Collaboration diagram for murasaki::I2cSlaveStrategy:



#### Public Member Functions

- virtual `murasaki::I2cStatus Transmit` (const uint8\_t \*tx\_data, unsigned int tx\_size, unsigned int \*transferred\_count=nullptr, unsigned int timeout\_ms=`murasaki::kwmsIndefinitely`)=0  
*Thread safe, synchronous transmission over I2C.*
- virtual `murasaki::I2cStatus Receive` (uint8\_t \*rx\_data, unsigned int rx\_size, unsigned int \*transferred\_count=nullptr, unsigned int timeout\_ms=`murasaki::kwmsIndefinitely`)=0  
*Thread safe, synchronous receiving over I2C.*

#### Additional Inherited Members

##### 7.24.1 Detailed Description

This class is a prototype of the I2C slave peripheral.

This prototype assumes the derived class will transmit/receive data in the task context on RTOS. And these member functions should be synchronous. That mean until the transmit / receive terminates, both method doesn't return.

These two callback member functions are prepared to sync with the interrupt, which tells the end of Transmit/Receive.

##### 7.24.2 Member Function Documentation

- 7.24.2.1 virtual `murasaki::I2cStatus murasaki::I2cSlaveStrategy::Receive` ( uint8\_t \* rx\_data, unsigned int rx\_size, unsigned int \* transferred\_count = nullptr, unsigned int timeout\_ms = `murasaki::kwmsIndefinitely` )  
 [pure virtual]

#### Parameters

<code>rx_data</code>	Data array to transmit.
<code>rx_size</code>	Data counts[bytes] to transmit.
<code>transferred_count</code>	the count of the bytes transferred during the API execution.
<code>timeout_ms</code>	Time ou [mS]. By default, there is not timeout.

**Returns**

Result of the processing

This member function is programmed to run in the task context of RTOS. This function should be internally exclusive between multiple task access. In other words, it should be thread-safe.

Implemented in [murasaki::I2cSlave](#).

```
7.24.2.2 virtual murasaki::I2cStatus murasaki::I2cSlaveStrategy::Transmit ( const uint8_t * tx_data, unsigned int tx_size,
    unsigned int * transfered_count = nullptr, unsigned int timeout_ms = murasaki::kwmsIndefinitely )
    [pure virtual]
```

**Parameters**

<i>tx_data</i>	Data array to transmit.
<i>tx_size</i>	Data counts[bytes] to transmit.
<i>transfered_count</i>	the count of the bytes transferred during the API execution.
<i>timeout_ms</i>	Time ou [mS]. By default, there is not timeout.

**Returns**

Result of the processing

This member function is programmed to run in the task context of RTOS. This function should be internally exclusive between multiple task access. In other words, it should be thread-safe.

Implemented in [murasaki::I2cSlave](#).

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

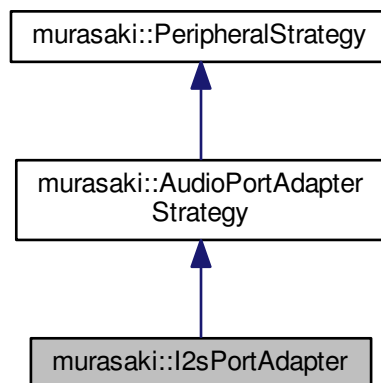
- [i2cslavestrategy.hpp](#)

**7.25 murasaki::I2sPortAdapter Class Reference**

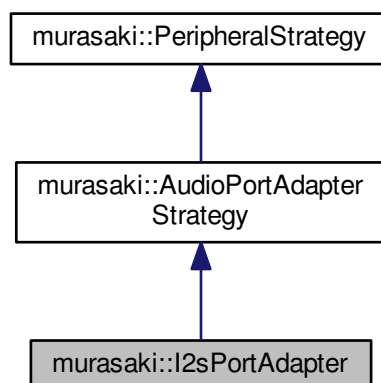
Adapter as I2S audio port.

```
#include <i2sportadapter.hpp>
```

Inheritance diagram for murasaki::I2sPortAdapter:



Collaboration diagram for murasaki::I2sPortAdapter:



#### Public Member Functions

- [I2sPortAdapter](#) (I2S\_HandleTypeDef \*tx\_peripheral, I2S\_HandleTypeDef \*rx\_peripheral)  
*Constructor.*
- virtual void [StartTransferTx](#) (uint8\_t \*tx\_buffer, unsigned int channel\_len)  
*Kick start routine to start the TX DMA transfer.*
- virtual void [StartTransferRx](#) (uint8\_t \*rx\_buffer, unsigned int channel\_len)  
*Kick start routine to start the RX DMA transfer.*
- virtual unsigned int [GetNumberOfDMAPhase](#) ()  
*Return how many DMA phase is implemented.*

- virtual unsigned int [GetNumberOfChannelsTx](#) ()  
*Return how many channels are in the transfer.*
- virtual unsigned int [GetSampleShiftSizeTx](#) ()  
*Return the bit count to shift to make the DMA data to the left align in the TX I2S frame.*
- virtual unsigned int [GetSampleWordSizeTx](#) ()  
*Return the size of the one sample on memory for Tx channel.*
- virtual unsigned int [GetNumberOfChannelsRx](#) ()  
*Return how many channels are in the transfer.*
- virtual unsigned int [GetSampleShiftSizeRx](#) ()  
*Return the bit count to shift to make the DMA data to right align from the left aligned RX I2S frame.*
- virtual unsigned int [GetSampleWordSizeRx](#) ()  
*Return the size of the one sample on memory for Rx channel.*
- virtual bool [HandleError](#) (void \*ptr)  
*Handling error report of device.*
- virtual bool [Match](#) (void \*peripheral\_handle)
- virtual void \* [GetPeripheralHandle](#) ()  
*pass the raw peripheral handler*
- virtual bool [IsInt16SwapRequired](#) ()  
*Display half word swap is required. .*

#### Additional Inherited Members

#### 7.25.1 Detailed Description

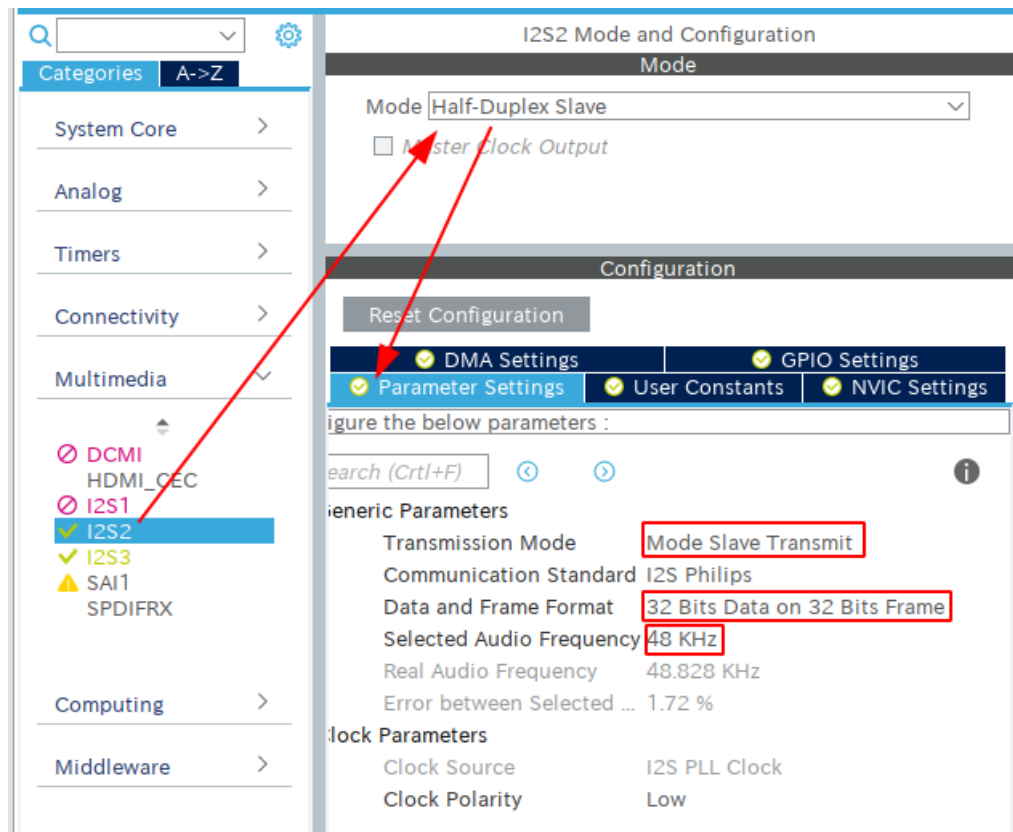
This class is a dedicated adapter class for the [murasaki::DuplexAudio](#). The [DuplexAudio](#) class can handle audio through the I2S port by using this adapter.

Caution : This class doesn't support the STM32H7 series duplex I2S.

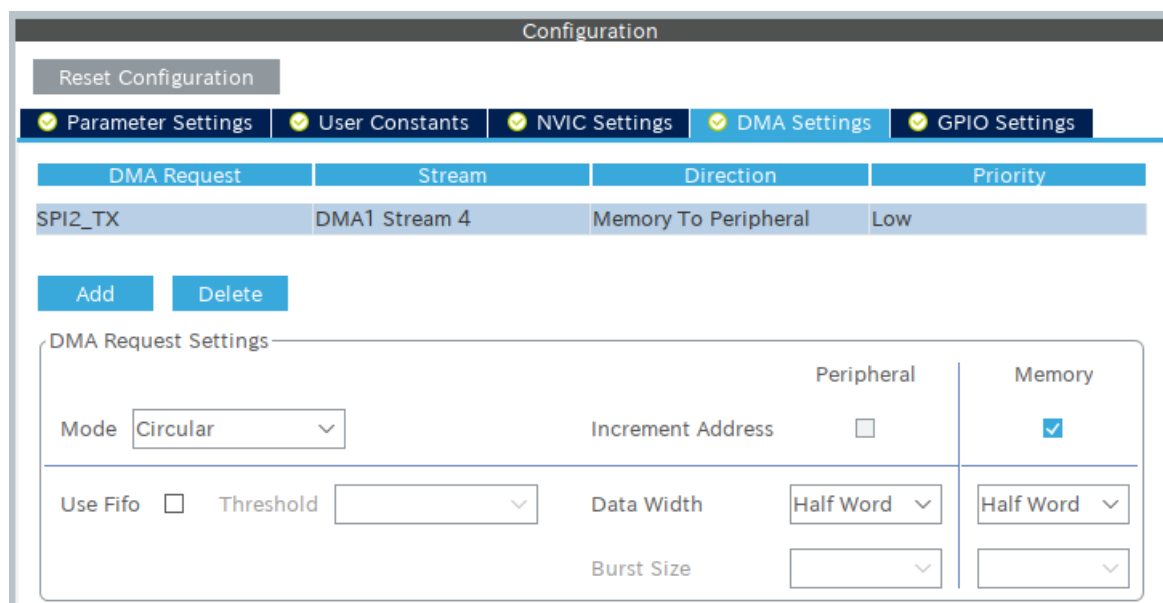
#### Configuration

To configure the I2S peripheral,

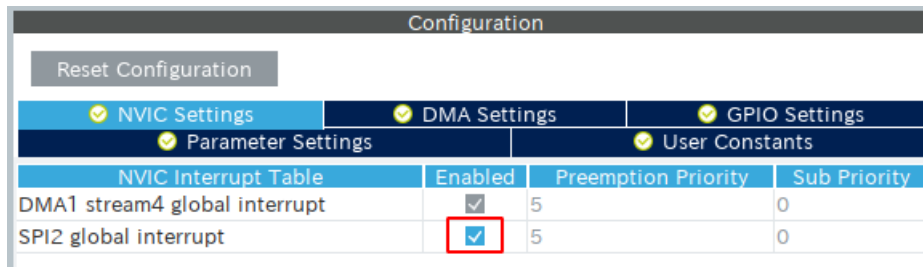
- Set the working mode as half-duplex slave or half-duplex master.
- Select the transmitter or receiver
- Select the Data and Frame Format. Anyone of four formats are allowed.
- Select appropriate audio frequency (Fs), if the peripheral is master.



The configuration of DMA is tricky. The mode must always be "Circular," and DMA size must always be "half word" for both memory and peripheral.



And then enable the interrupt if the peripheral is a receiver. Enabling the interrupt for a transmitter does not affect.



The following table summarizes the data size in I2S signal frame vs. the configurator settings. You must configure the DMA as a circular mode.

I2S Data Size	I2S Data and Frame Format	I2S DMA Data Size
16 bits	16bits data and 16bits frame	Half Word
16 bits	16bits data and 32bits frame	Half Word
24 bits	24bits data and 24bits frame	Half Word
32 bits	32bits data and 32bits frame	Half Word

### Creating a peripheral object

To create an object, pass the handles of the I2S port as a parameter. In the following example, the hi2s2 and hi2s3 are configured as TX and RX, respectively. And both are generated by CubeIDE.

The created [I2sPortAdapter](#) object has to be passed to the [DuplexAudio](#) constructor.

```
audio_port = new murasaki::I2sPortAdapter(&hi2s2, &hi2s3);
audio = new murasaki::DuplexAudio( audio_port );
```

## 7.25.2 Constructor & Destructor Documentation

### 7.25.2.1 murasaki::I2sPortAdapter::I2sPortAdapter ( I2S\_HandleTypeDef \* tx\_peripheral, I2S\_HandleTypeDef \* rx\_peripheral )

#### Parameters

<i>tx_peripheral</i>	I2S_HandleTypeDef type peripheral for TX. This is defined in <a href="#">main.c</a> .
<i>rx_peripheral</i>	I2S_HandleTypeDef type peripheral for RX. This is defined in <a href="#">main.c</a> .

This constructor function receives the handle of the I2S block peripherals.

This class assumes one is the TX, and the other is RX. In the case of a programmer use I2S as simplex audio, the unused block must be passed as nullptr.

## 7.25.3 Member Function Documentation

### 7.25.3.1 virtual unsigned int murasaki::I2sPortAdapter::GetNumberOfChannelsRx ( ) [inline],[virtual]

#### Returns

always 2

Implements [murasaki::AudioPortAdapterStrategy](#).



7.25.3.2 virtual unsigned int murasaki::I2sPortAdapter::GetNumberOfChannelsTx ( ) [inline],[virtual]

#### Returns

always 2

Implements [murasaki::AudioPortAdapterStrategy](#).

7.25.3.3 virtual unsigned int murasaki::I2sPortAdapter::GetNumberOfDMAPhase ( ) [inline],[virtual]

#### Returns

Always return 2 for STM32 I2S, because the cyclic DMA has halfway and complete interrupt.

Implements [murasaki::AudioPortAdapterStrategy](#).

7.25.3.4 void \* murasaki::I2sPortAdapter::GetPeripheralHandle ( ) [virtual]

#### Returns

pointer to the raw peripheral handler hidden in a class.

Implements [murasaki::AudioPortAdapterStrategy](#).

7.25.3.5 virtual unsigned int murasaki::I2sPortAdapter::GetSampleShiftSizeRx ( ) [inline],[virtual]

#### Returns

0 The unit is [bit]

Return the bit count to shift to make the DMA data to right align from the left aligned RX I2S frame.

#### Returns

0 The unit is [bit]

This member function is needed because of the mismatch in the DMA buffer data format and I2S format.

Let's assume the 24bit data I2S format. Some peripheral place the data as right-aligned in 32bit DMA data ( as integer ), some peripheral places the data as left-aligned in 32bit DMA data ( as a fixed point ).

The audio framework ( [DuplexAudio](#) ) compensates for this kind of mismatch. The This member function returns how many bits have to be shifted to the right in RX. If peripheral requires left align format, this function should return 0. The STM32 I2S DMA format is left aligned. So, always return 0.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.25.3.6** `virtual unsigned int murasaki::I2sPortAdapter::GetSampleShiftSizeTx ( ) [inline],[virtual]`

#### Returns

Always 0.

This member function is needed because of the mismatch in the DMA buffer data format and I2S format.

Let's assume the 24bit data I2S format. Some peripheral place the data as right-aligned in 32bit DMA data ( as integer ), some peripheral places the data as left-aligned in 32bit DMA data ( as a fixed point ).

The audio framework ( [DuplexAudio](#) ) compensates for this kind of mismatch. This member function returns how many bits have to be shifted to the left in TX.

If peripheral requires left align format, this function should return 0.

The STM32 I2S DMA format is left aligned. So, always return 0. Kick start routine to start the RX DMA transfer.

This routine must be implemented by the derived class. The task of this routine is to kick the first DMA transfer. In this class, we assume DMA continuously transfers on the circular buffer once after it starts.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.25.3.7** `unsigned int murasaki::I2sPortAdapter::GetSampleWordSizeRx ( ) [virtual]`

#### Returns

2 or 4. The unit is [Byte]

This function returns the size of the word which should be allocated on the memory.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.25.3.8** `unsigned int murasaki::I2sPortAdapter::GetSampleWordSizeTx ( ) [virtual]`

#### Returns

2 or 4. The unit is [Byte]

This function returns the size of the word which should be allocated on the memory.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.25.3.9** `bool murasaki::I2sPortAdapter::HandleError ( void * ptr ) [virtual]`

#### Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a device control
------------	---

#### Returns

true if ptr matches with device and handle the error. false if ptr doesn't match A member function to detect error.

The error handling is depend on the implementation.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.25.3.10** `virtual bool murasaki::I2sPortAdapter::IsInt16SwapRequired ( ) [inline],[virtual]`

Returns

Always ture.

Display whether the half word (int16\_t) swap is required or not.

I2S DMA requires the half word swap inside word. Thus, always returns true.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.25.3.11** `bool murasaki::I2sPortAdapter::Match ( void * peripheral_handle ) [virtual]`

Check if peripheral handle matched with given handle.

Parameters

<i>peripheral_handle</i>	
--------------------------	--

Returns

true if match, false if not match.

The SaiAudioAdapter type has two peripheral. TX and RX. This function checks RX paripheral and return the result of checking with this given pointer. That means, if RX is not nullptr, TX is not checked.

TX is checked only when, RX is nullptr.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.25.3.12** `void murasaki::I2sPortAdapter::StartTransferRx ( uint8_t * rx_buffer, unsigned int channel_len ) [virtual]`

This routine must be implemented by the derived class. The task of this routine is to kick the first DMA transfer. In this class, we assume DMA continuously transfers on the circular buffer once after it starts.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.25.3.13** `void murasaki::I2sPortAdapter::StartTransferTx ( uint8_t * tx_buffer, unsigned int channel_len ) [virtual]`

This routine must be implemented by the derived class. The task of this routine is to kick the first DMA transfer. In this class, we assume DMA continuously transfers on the circular buffer once after it starts.

Implements [murasaki::AudioPortAdapterStrategy](#).

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

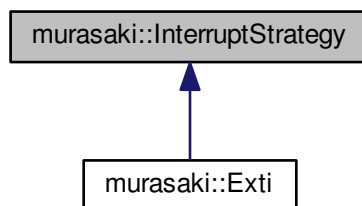
- [i2sportadapter.hpp](#)
- [i2sportadapter.cpp](#)

## 7.26 murasaki::InterruptStrategy Class Reference

Abstract interrupt class.

```
#include <interruptstrategy.hpp>
```

Inheritance diagram for murasaki::InterruptStrategy:



### Public Member Functions

- virtual void `Enable` ()=0  
*Enable interrupt.*
- virtual void `Disable` ()=0  
*Disable interrupt.*
- virtual `murasaki::InterruptStatus Wait` (unsigned int timeout=`murasaki::kwmsIndefinitely`)=0
- virtual bool `Release` (unsigned int line)=0
- virtual bool `Match` (unsigned int line)=0

#### 7.26.1 Detailed Description

This is a mother of the interrupt classes. The interrupt class provides following functionalities :

- Enable/ Disable interrupt line by line.
- Synchronization between interrupt and task.

#### 7.26.2 Member Function Documentation

##### 7.26.2.1 virtual bool murasaki::InterruptStrategy::Match ( unsigned int *line* ) [pure virtual]

Check whether given line is handled by this object.

#### Parameters

<i>line</i>	The identifier of the interrupt
-------------	---------------------------------

**Returns**

true if line is matched with this EXTI. false if not matched.

Implemented in [murasaki::Exti](#).

**7.26.2.2 virtual bool murasaki::InterruptStrategy::Release ( unsigned int *line* ) [pure virtual]**

Release the waiting task

**Parameters**

<i>line</i>	The identifier of the interrupt
-------------	---------------------------------

**Returns**

true if line is matched with this EXTI and release. false if not matched.

Implemented in [murasaki::Exti](#).

**7.26.2.3 virtual murasaki::InterruptStatus murasaki::InterruptStrategy::Wait ( unsigned int *timeout* = murasaki::kwmnsIndefinitely ) [pure virtual]**

Wait for an interrupt from task.

**Parameters**

<i>timeout</i>	time out parameter
----------------	--------------------

**Returns**

kisOK : Released by interrupt. kisTimeOut : released by timeout.

Implemented in [murasaki::Exti](#).

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

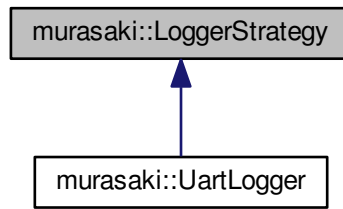
- [interruptstrategy.hpp](#)

**7.27 murasaki::LoggerStrategy Class Reference**

Abstract class for logging.

```
#include <loggerstrategy.hpp>
```

Inheritance diagram for `murasaki::LoggerStrategy`:



### Public Member Functions

- virtual `~LoggerStrategy()`  
*Destructor.*
- virtual void `putMessage` (char message[], unsigned int size)=0  
*Message output member function.*
- virtual char `getCharacter` ()=0  
*Character input member function.*
- virtual void `DoPostMortem` (void \*debugger\_fifo)  
*Start post mortem process.*

### Static Protected Member Functions

- static void \* `GetPeripheralHandle` (murasaki::PeripheralStrategy \*peripheral)  
*This special method helps derived loggers. The loggers can access the raw device, in case of the post mortem processing.*

#### 7.27.1 Detailed Description

A generic class to serve a logging function. This class is designed to pass to the `murasaki::Debugger`.

As a service class to Debug. This class's two member functions ( `putMessage()` and `getCharacter()` ) have to be able to run in the task context. Both member functions also have to be the blocking and synchronous function.

#### 7.27.2 Constructor & Destructor Documentation

##### 7.27.2.1 virtual murasaki::LoggerStrategy::~~LoggerStrategy ( ) [inline],[virtual]

Do nothing here. Declared to enforce the derived class's constructor as "virtual".

#### 7.27.3 Member Function Documentation

##### 7.27.3.1 virtual void murasaki::LoggerStrategy::DoPostMortem ( void \* debugger\_fifo ) [inline],[virtual]

## Parameters

<i>debugger_fifo</i>	Pointer to the <a href="#">DebuggerFifo</a> class object. This is declared as void to avoid the include confusion. This member function read the data in given FIFO, and then do the auto history.
----------------------	--

By default this is not implemented. But in case user implements a method, it should call the `Debugger::SetPostMortem()` internally.

Reimplemented in [murasaki::UartLogger](#).

### 7.27.3.2 virtual char murasaki::LoggerStrategy::getCharacter ( ) [pure virtual]

## Returns

A character from input is returned.

This function is considered as blocking and synchronous. That mean, the function will wait for any user input forever.

Implemented in [murasaki::UartLogger](#).

### 7.27.3.3 virtual void murasaki::LoggerStrategy::putMessage ( char *message*[], unsigned int *size* ) [pure virtual]

## Parameters

<i>message</i>	Non null terminated character array. This data is stored or output to the logger.
<i>size</i>	Byte length of the message parameter of the putMessage member function.

This function is considered as asynchronous and blocking. That mean, it will not wait until data is stored to the storage or output.

For example, if there is not room in FIFO anymore, this member function will just return without putting data.

Implemented in [murasaki::UartLogger](#).

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

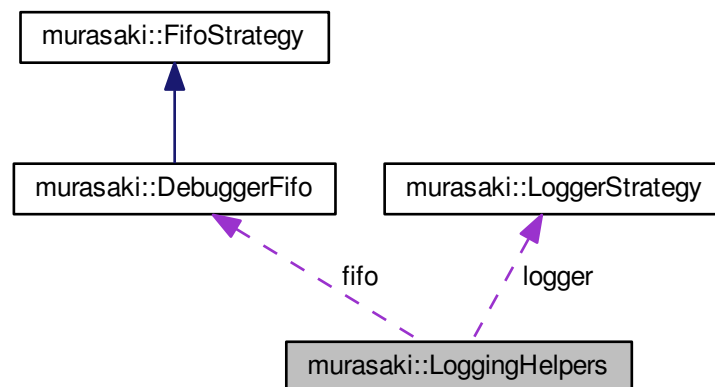
- [loggerstrategy.hpp](#)

## 7.28 murasaki::LoggingHelpers Struct Reference

A stracture to engroup the logging tools.

```
#include <debuggerfifo.hpp>
```

Collaboration diagram for `murasaki::LoggingHelpers`:



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

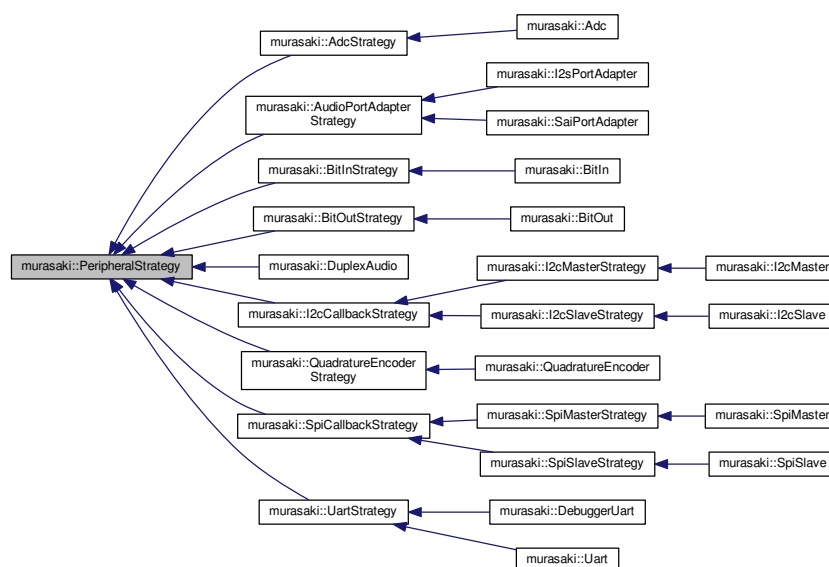
- [debuggerfifo.hpp](#)

## 7.29 murasaki::PeripheralStrategy Class Reference

Mother of all peripheral class.

```
#include <peripheralstrategy.hpp>
```

Inheritance diagram for `murasaki::PeripheralStrategy`:





## Public Member Functions

- virtual [~PeripheralStrategy](#) ()  
*destructor*
- virtual bool [Match](#) (void \*peripheral\_handle)

## Protected Member Functions

- virtual void \* [GetPeripheralHandle](#) ()=0  
*pass the raw peripheral handler*

## 7.29.1 Detailed Description

This class provides the [GetPeripheralHandle\(\)](#) member function as a common stub for the debugging logger. The loggers sometimes refers the raw peripheral to respond to the post mortem situation. By using class, programmer can pass the raw peripheral handler to loggers, while keep it hidden from the application.

## 7.29.2 Member Function Documentation

7.29.2.1 virtual void\* murasaki::PeripheralStrategy::GetPeripheralHandle ( ) [protected],[pure virtual]

## Returns

pointer to the raw peripheral handler hidden in a class.

Implemented in [murasaki::DuplexAudio](#), [murasaki::I2sPortAdapter](#), [murasaki::SaiPortAdapter](#), [murasaki::AudioPortAdapterStrategy](#), [murasaki::Adc](#), [murasaki::BitOut](#), and [murasaki::BitIn](#).

7.29.2.2 virtual bool murasaki::PeripheralStrategy::Match ( void \* *peripheral\_handle* ) [inline],[virtual]

Check if peripheral handle matched with given handle.

## Parameters

<i>peripheral_handle</i>	
--------------------------	--

## Returns

true if match, false if not match.

Reimplemented in [murasaki::DuplexAudio](#), [murasaki::I2sPortAdapter](#), [murasaki::SaiPortAdapter](#), and [murasaki::AudioPortAdapterStrategy](#).

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

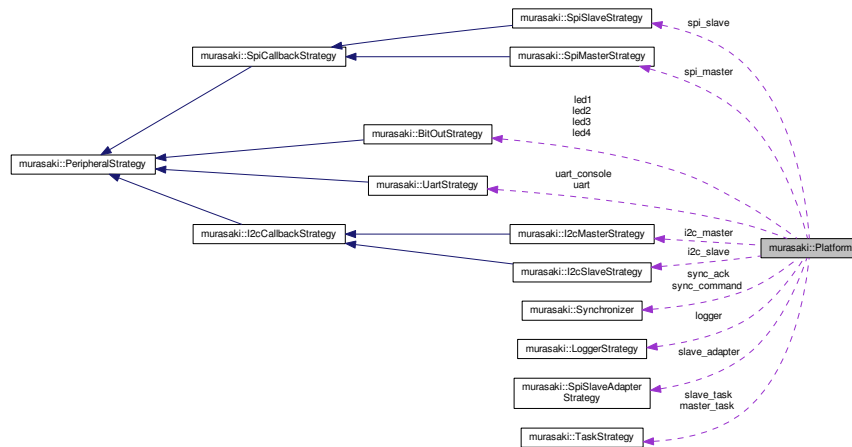
- [peripheralstrategy.hpp](#)

### 7.30 murasaki::Platform Struct Reference

Custom aggregation struct for user platform.

```
#include <platform_defs.hpp>
```

Collaboration diagram for murasaki::Platform:



#### Public Attributes

- [UartStrategy](#) \* [uart\\_console](#)  
*UART wrapping class object for debugging.*
- [LoggerStrategy](#) \* [logger](#)  
*logging class object for debugger*
- [BitOutStrategy](#) \* [led1](#)  
*GP out under test.*
- [BitOutStrategy](#) \* [led2](#)  
*GP out under test.*
- [BitOutStrategy](#) \* [led3](#)  
*GP out under test.*
- [BitOutStrategy](#) \* [led4](#)  
*GP out under test.*
- [TaskStrategy](#) \* [master\\_task](#)  
*Task under test.*
- [TaskStrategy](#) \* [slave\\_task](#)  
*Task under test.*
- [I2cMasterStrategy](#) \* [i2c\\_master](#)  
*I2C Master under test.*
- [I2cSlaveStrategy](#) \* [i2c\\_slave](#)  
*I2C Slave under test.*
- [UartStrategy](#) \* [uart](#)  
*UART under test.*
- [SpiMasterStrategy](#) \* [spi\\_master](#)  
*SPI Master under test.*
- [SpiSlaveStrategy](#) \* [spi\\_slave](#)  
*SPI Slave under test.*

### 7.30.1 Detailed Description

A collection of the peripheral / MPU control variable.

This is a custom struct. Programmer can change this struct as suitable to the hardware and software. But `debugger_` member variable have to be left untouched.

In the run time, the `debugger_` variable have to be initialized by appropriate [murasaki::Debugger](#) class instance.

See [murasaki::platform](#)

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

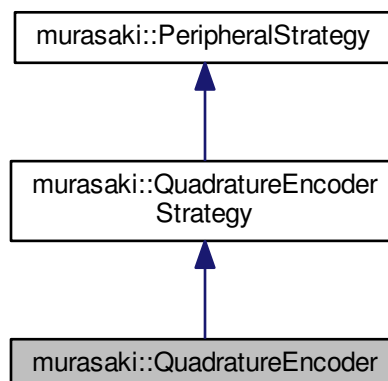
- [platform\\_defs.hpp](#)

## 7.31 **murasaki::QuadratureEncoder Class Reference**

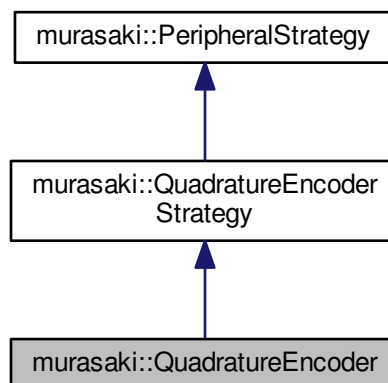
Quadrature Encoder class.

```
#include <quadratureencoder.hpp>
```

Inheritance diagram for `murasaki::QuadratureEncoder`:



Collaboration diagram for `murasaki::QuadratureEncoder`:



#### Public Member Functions

- `QuadratureEncoder` ()=delete
- `QuadratureEncoder` (TIM\_HandleTypeDef \*htim)  
*Constructor.*
- virtual unsigned int `Get` ()  
*Get the internal counter value.*
- virtual void `Set` (unsigned int value)  
*Set the internal counter.*

#### Additional Inherited Members

##### 7.31.1 Detailed Description

Dedicated class for the STM32 Timer encoder. The STM32 timer encoder can accept the 2 signal from the quadrature pulse encoder. This timer encoder allow to up / down the counter by CW/CCW rotation. Basic parameter have to be set by CubeIDE/CubeMX

##### 7.31.2 Constructor & Destructor Documentation

###### 7.31.2.1 `murasaki::QuadratureEncoder::QuadratureEncoder ( )` [delete]

Suppress the default constructor.

###### 7.31.2.2 `murasaki::QuadratureEncoder::QuadratureEncoder ( TIM_HandleTypeDef * htim )`

## Parameters

<code>htim</code>	The pointer to the peripheral control structure.
-------------------	--

The given parameter is stored to the `peripheral_` variable.

## 7.31.3 Member Function Documentation

7.31.3.1 `unsigned int murasaki::QuadratureEncoder::Get ( void ) [virtual]`

## Returns

The value of the internal counter.

Return the encoder internal value.

Implements [murasaki::QuadratureEncoderStrategy](#).

7.31.3.2 `void murasaki::QuadratureEncoder::Set ( unsigned int value ) [virtual]`

## Parameters

<code>value</code>	The value to set.
--------------------	-------------------

The encoder internal counter is set to the value.

Implements [murasaki::QuadratureEncoderStrategy](#).

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

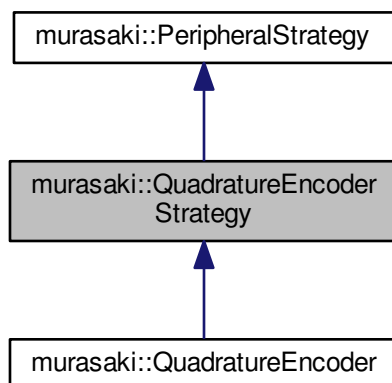
- [quadratureencoder.hpp](#)
- [quadratureencoder.cpp](#)

7.32 `murasaki::QuadratureEncoderStrategy` Class Reference

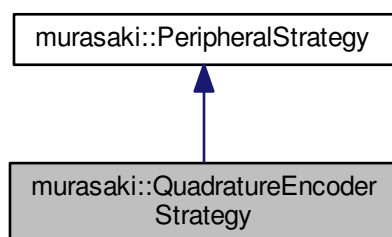
Strategy class for the quadrature encoder.

```
#include <quadratureencoderstrategy.hpp>
```

Inheritance diagram for `murasaki::QuadratureEncoderStrategy`:



Collaboration diagram for `murasaki::QuadratureEncoderStrategy`:



#### Public Member Functions

- virtual unsigned int `Get` ()=0  
*Get the internal counter value.*
- virtual void `Set` (unsigned int value)=0  
*Set the internal counter.*

#### Additional Inherited Members

##### 7.32.1 Detailed Description

The strategy class of the quadrature encoder peripheral. The inherited class will control the encoder.

## 7.32.2 Member Function Documentation

7.32.2.1 `virtual unsigned int murasaki::QuadratureEncoderStrategy::Get ( ) [pure virtual]`

## Returns

The value of the internal counter.

Return the encoder internal value.

Implemented in [murasaki::QuadratureEncoder](#).

7.32.2.2 `virtual void murasaki::QuadratureEncoderStrategy::Set ( unsigned int value ) [pure virtual]`

## Parameters

<i>value</i>	The value to set.
--------------	-------------------

The encoder internal counter is set to the value.

Implemented in [murasaki::QuadratureEncoder](#).

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

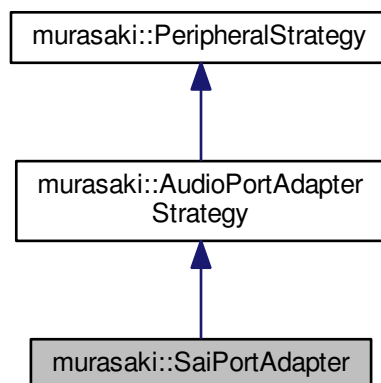
- [quadratureencoderstrategy.hpp](#)

## 7.33 murasaki::SaiPortAdapter Class Reference

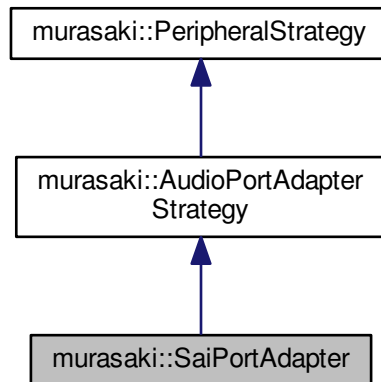
Adapter as SAI audio port.

```
#include <saiportadapter.hpp>
```

Inheritance diagram for murasaki::SaiPortAdapter:



Collaboration diagram for `murasaki::SaiPortAdapter`:



#### Public Member Functions

- [SaiPortAdapter](#) (`SAI_HandleTypeDef *tx_peripheral, SAI_HandleTypeDef *rx_peripheral`)  
*Constructor.*
- virtual void [StartTransferTx](#) (`uint8_t *tx_buffer, unsigned int channel_len`)  
*Kick start routine to start the TX DMA transfer.*
- virtual void [StartTransferRx](#) (`uint8_t *rx_buffer, unsigned int channel_len`)  
*Kick start routine to start the RX DMA transfer.*
- virtual unsigned int [GetNumberOfDMAPhase](#) ()  
*Return how many DMA phase is implemented.*
- virtual unsigned int [GetNumberOfChannelsTx](#) ()  
*Return how many channels are in the transfer.*
- virtual unsigned int [GetSampleShiftSizeTx](#) ()  
*Return the bit count to shift to make the DMA data to right align in TX I2S frame.*
- virtual unsigned int [GetSampleWordSizeTx](#) ()  
*Return the size of the one sample on memory for Tx channel.*
- virtual unsigned int [GetNumberOfChannelsRx](#) ()  
*Return how many channels are in the transfer.*
- virtual unsigned int [GetSampleShiftSizeRx](#) ()  
*Return the bit count to shift to make the DMA data to right align in RX I2S frame.*
- virtual unsigned int [GetSampleWordSizeRx](#) ()  
*Return the size of the one sample on memory for Rx channel.*
- virtual bool [HandleError](#) (`void *ptr`)  
*Handling error report of device.*
- virtual bool [Match](#) (`void *peripheral_handle`)
- virtual void \* [GetPeripheralHandle](#) ()  
*pass the raw peripheral handler*
- virtual bool [IsInt16SwapRequired](#) ()  
*Display half word swap is required. .*



## Additional Inherited Members

### 7.33.1 Detailed Description

Dedicated adapter for the [murasaki::DuplexAudio](#). By passing this adapter, the [DuplexAudio](#) class can handle audio through the SAI port.

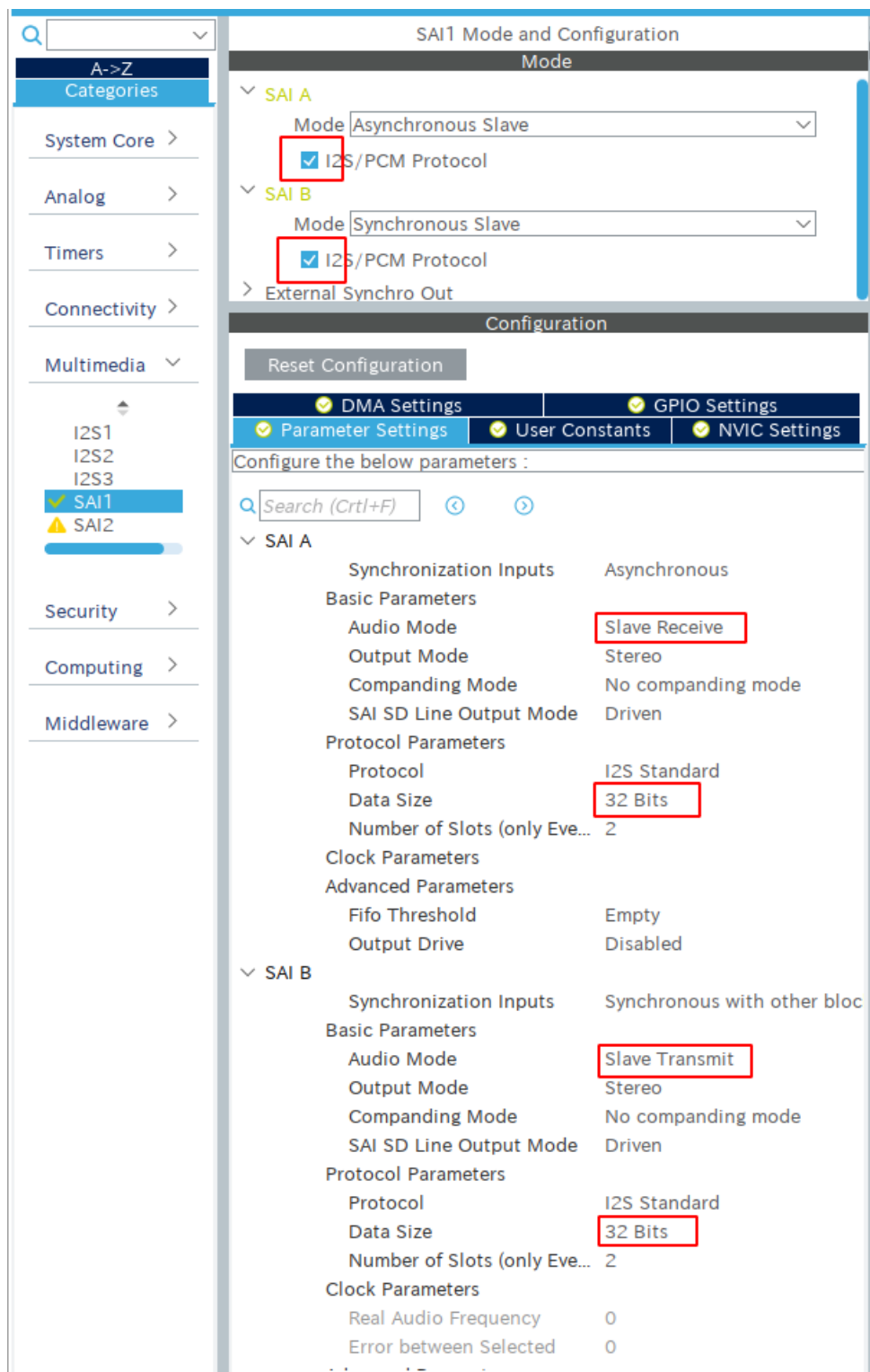
Caution : The size of the data in SAI and the width of the data in DMA must be aligned. This is responsibility of the programmer. The mis-aligned configuration gives broken audio.

Caution : This class doesn't support the STM32H7 series duplex I2S.

## Configuration

To configure the SAI peripheral,

- Set the mode as appropriate to your circuit
- Select the Audio Mode transmitter or receiver
- Select the Data size. Any one of four formats are allowed.
- Select appropriate audio frequency (Fs), if the peripheral is master.



The configuration of DMA is tricky. Set the DMA based on the table below.

Configuration

Reset Configuration

Parameter Settings User Constants NVIC Settings **DMA Settings** GPIO Settings

DMA Request	Stream	Direction	Priority
SAI1_A	DMA2 Stream 1	Peripheral To Memory	Low
SAI1_B	DMA2 Stream 4	Memory To Peripheral	Low

Add Delete

DMA Request Settings

Mode: Circular

Increment Address: ☐

Use Fifo: ☐ Threshold:

Data Width: Word Burst Size:

Peripheral: ☐ Memory: ☒

And then, enable the interrupt .

Configuration

Reset Configuration

Parameter Settings User Constants **NVIC Settings** DMA Settings GPIO Settings

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
DMA2 stream1 global interrupt	<input checked="" type="checkbox"/>	5	0
DMA2 stream4 global interrupt	<input checked="" type="checkbox"/>	5	0
SAI1 global interrupt	<input checked="" type="checkbox"/>	5	0

Following table summarizes the data size in I2S signal frame vs the configurator settings. DMA must be configured as circular mode.

I2S Data Size	SAI Data Size	SAI DMA Data Size
16 bits	16 bits	Half Word
16 bits	16 bits Extended	Half Word
24 bits	24 bits	Word
32 bits	32 bits	Word

### Creating a peripheral object

To create an object, pass the handles of the SAI port as parameter. In the following example, the `hsai_BlockA1` and `hsai_BlockB1` are configured as TX and RX, respectively. And both are generated by CubeIDE.

The created [SaiPortAdapter](#) object have to be passed to [DuplexAudio](#) constructor.

```
audio_port = new murasaki::SaiPortAdapter(&hsai_BlockA1, &hsai_BlockB1);
audio = new murasaki::DuplexAudio( audio_port );
```

### 7.33.2 Constructor & Destructor Documentation

#### 7.33.2.1 murasaki::SaiPortAdapter::SaiPortAdapter ( SAI\_HandleTypeDef \* tx\_peripheral, SAI\_HandleTypeDef \* rx\_peripheral )

## Parameters

<i>tx_peripheral</i>	SAI_HandleTypeDef type peripheral for TX. This is defined in <a href="#">main.c</a> .
<i>rx_peripheral</i>	SAI_HandleTypeDef type peripheral for RX. This is defined in <a href="#">main.c</a> .

Receives handle of the SAI block peripherals.

SAI has two block internally. This class assumes one is the TX and the other is RX. In case of a programmer use SAI as simplex audio, the unused block must be passed as nullptr.

## 7.33.3 Member Function Documentation

7.33.3.1 `unsigned int murasaki::SaiPortAdapter::GetNumberOfChannelsRx ( ) [virtual]`

## Returns

1 for Mono, 2 for stereo, 3... for multi-channel.

Implements [murasaki::AudioPortAdapterStrategy](#).

7.33.3.2 `unsigned int murasaki::SaiPortAdapter::GetNumberOfChannelsTx ( ) [virtual]`

## Returns

1 for Mono, 2 for stereo, 3... for multi-channel.

Implements [murasaki::AudioPortAdapterStrategy](#).

7.33.3.3 `virtual unsigned int murasaki::SaiPortAdapter::GetNumberOfDMAPhase ( ) [inline],[virtual]`

## Returns

Always return 2 for STM32 SAI, because the cyclic DMA has halfway and complete interrupt.

Implements [murasaki::AudioPortAdapterStrategy](#).

7.33.3.4 `void * murasaki::SaiPortAdapter::GetPeripheralHandle ( ) [virtual]`

## Returns

pointer to the raw peripheral handler hidden in a class.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.33.3.5** unsigned int murasaki::SaiPortAdapter::GetSampleShiftSizeRx ( ) [virtual]**Returns**

0 The unit is [bit]

This is needed because of the mismatch in the DMA buffer data format and I2S format.

Let's assume the 24bit data I2S format. Some peripheral place the data as right aligned in 32bit DMA data ( as integer ), some peripheral places the data as left aligned in 32bit DMA data ( as fixed point ).

This kind of the mismatch will be aligned by audio frame work. This member function returns how many bits have to be shifted to left in RX.

If peripheral requires left align format, this function should return 0.

The STM32 I2S DMA format is left aligned. So, always return 0.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.33.3.6** unsigned int murasaki::SaiPortAdapter::GetSampleShiftSizeTx ( ) [virtual]**Returns**

Always 0.

This is needed because of the mismatch in the DMA buffer data format and I2S format.

Let's assume the 24bit data I2S format. Some peripheral place the data as right aligned in 32bit DMA data ( as integer ), some peripheral places the data as left aligned in 32bit DMA data ( as fixed point ).

This kind of the mismatch will be aligned by audio frame work. This member function returns how many bits have to be shifted to right in TX.

If peripheral requires left align format, this function should return 0.

The STM32 SAI DMA format is right aligned.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.33.3.7** unsigned int murasaki::SaiPortAdapter::GetSampleWordSizeRx ( ) [virtual]**Returns**

2 or 4. The unit is [Byte]

This function returns the size of the word which should be allocated on the memory.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.33.3.8** unsigned int murasaki::SaiPortAdapter::GetSampleWordSizeTx ( ) [virtual]**Returns**

2 or 4. The unit is [Byte]

This function returns the size of the word which should be allocated on the memory.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.33.3.9** bool murasaki::SaiPortAdapter::HandleError ( void \* ptr ) [virtual]

**Parameters**

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a device control
------------	---

**Returns**

true if ptr matches with device and handle the error. false if ptr doesn't match A member function to detect error.

The error handling is depend on the implementation.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.33.3.10** `bool murasaki::SaiPortAdapter::IsInt16SwapRequired ( ) [virtual]`

**Returns**

Always false.

Display whether the half word (int16\_t) swap is required or not.

SAI doesn't require the half word swap inside word. Thus, always returns false

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.33.3.11** `bool murasaki::SaiPortAdapter::Match ( void * peripheral_handle ) [virtual]`

Check if peripheral handle matched with given handle.

**Parameters**

<i>peripheral_handle</i>	
--------------------------	--

**Returns**

true if match, false if not match.

The SaiAudioAdapter type has two peripheral. TX and RX. This function checks RX paripheral and return with this value. That means, if RX is not nullptr, TX is not checked.

TX is checked only when, RX is nullptr.

Implements [murasaki::AudioPortAdapterStrategy](#).

**7.33.3.12** `void murasaki::SaiPortAdapter::StartTransferRx ( uint8_t * rx_buffer, unsigned int channel_len ) [virtual]`

This routine must be implemented by the derived class. The task of this routine is to kick the first DMA transfer. In this class, we assume DMA continuously transfer on the circular buffer once after it starts.

Implements [murasaki::AudioPortAdapterStrategy](#).

7.33.3.13 `void murasaki::SaiPortAdapter::StartTransferTx ( uint8_t* tx_buffer, unsigned int channel_len ) [virtual]`

This routine must be implemented by the derived class. The task of this routine is to kick the first DMA transfer. In this class, we assume DMA continuously transfer on the circular buffer once after it starts.

Implements [murasaki::AudioPortAdapterStrategy](#).

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

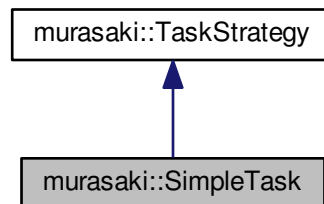
- [saiportadapter.hpp](#)
- [saiportadapter.cpp](#)

## 7.34 `murasaki::SimpleTask` Class Reference

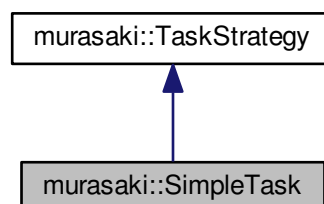
An easy to use task class.

```
#include <simpletask.hpp>
```

Inheritance diagram for `murasaki::SimpleTask`:



Collaboration diagram for `murasaki::SimpleTask`:



## Public Member Functions

- [SimpleTask](#) (const char \*task\_name, unsigned short stack\_depth, [murasaki::TaskPriority](#) task\_priority, const void \*task\_parameter, void(\*task\_body\_func)(const void \*))

*Ease to use task class.*

## Protected Member Functions

- virtual void [TaskBody](#) (const void \*ptr)

*Task member function.*

## Additional Inherited Members

### 7.34.1 Detailed Description

This is handy class to encapsulate the task creation without inheriting. A task can be created easy like :

```
// For demonstration of FreeRTOS task.
murasaki::platform.task1 = new murasaki::SimpleTask(
    "Master",
    256,
    (( configMAX_PRIORITIES > 1) ? 1 : 0),
    nullptr,
    &TaskBodyFunction
);
```

Then, task you can call [Start\(\)](#) member function to run.

```
murasaki::platform.task1->Start();
```

### 7.34.2 Constructor & Destructor Documentation

- 7.34.2.1** [murasaki::SimpleTask::SimpleTask](#) ( const char \* task\_name, unsigned short stack\_depth, [murasaki::TaskPriority](#) task\_priority, const void \* task\_parameter, void(\*) (const void \*) task\_body\_func )

#### Parameters

<i>task_name</i>	A name of task. This is relevant to the FreeRTOS's API manner.
<i>stack_depth</i>	Task stack size by byte.
<i>task_priority</i>	The task priority. Max priority is defined by configMAX_PRIORITIES in FreeRTOSConfig.h
<i>task_parameter</i>	A pointer to the parameter passed to task.
<i>task_body_func</i>	A pointer to the task body function.

Create an task object. Given parameters are stored internally. And then passed to the FreeRTOS API when task is started by [Start\(\)](#) member function.

A task parameter can be passed to task through the task\_parameter. This pointer is simply passed to the task body function without modification.



## 7.34.3 Member Function Documentation

7.34.3.1 void murasaki::SimpleTask::TaskBody ( const void \* *ptr* ) [protected], [virtual]

## Parameters

<i>ptr</i>	The task_parameter parameter of the constructor is passed to this parameter.
------------	--

This member function runs as task. In this function, the function passed thorough task\_body\_func parameter is invoked as actual task body.

Implements [murasaki::TaskStrategy](#).

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

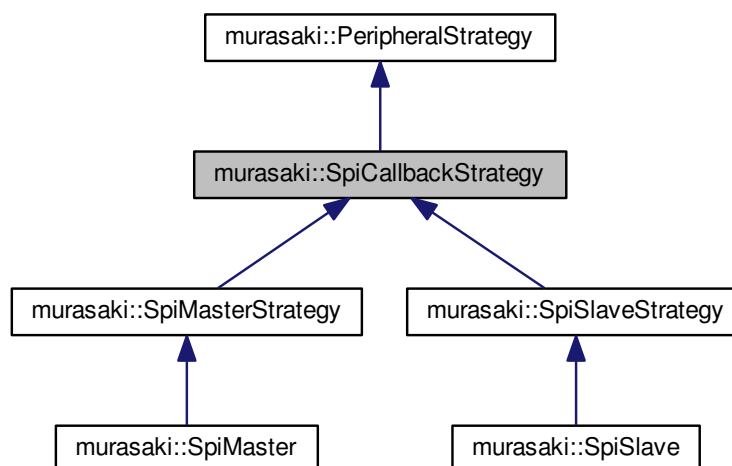
- [simpletask.hpp](#)
- [simpletask.cpp](#)

## 7.35 murasaki::SpiCallbackStrategy Class Reference

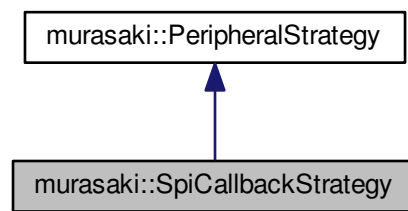
Root class of the SPI.

```
#include <spicallbackstrategy.hpp>
```

Inheritance diagram for murasaki::SpiCallbackStrategy:



Collaboration diagram for `murasaki::SpiCallbackStrategy`:



#### Public Member Functions

- virtual bool [TransmitAndReceiveCompleteCallback](#) (void \*ptr)=0  
*Callback to notify the end of transfer.*
- virtual bool [HandleError](#) (void \*ptr)=0  
*Handling error report of device.*

#### Additional Inherited Members

##### 7.35.1 Member Function Documentation

7.35.1.1 virtual bool `murasaki::SpiCallbackStrategy::HandleError ( void * ptr )` [pure virtual]

#### Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a device control
------------	---

#### Returns

true if ptr matches with device and handle the error. false if ptr doesn't match A member function to detect error.

The error handling is depend on the implementation.

Implemented in [murasaki::SpiSlave](#), and [murasaki::SpiMaster](#).

7.35.1.2 virtual bool `murasaki::SpiCallbackStrategy::TransmitAndReceiveCompleteCallback ( void * ptr )` [pure virtual]

#### Parameters

<i>ptr</i>	Pointer to the control object.
------------	--------------------------------

**Returns**

true if no error.

Implemented in [murasaki::SpiSlave](#), and [murasaki::SpiMaster](#).

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

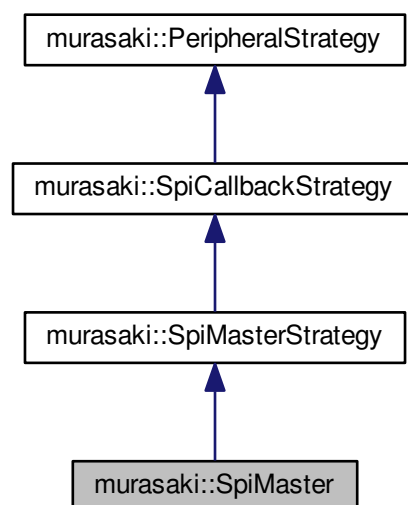
- [spicallbackstrategy.hpp](#)

**7.36 murasaki::SpiMaster Class Reference**

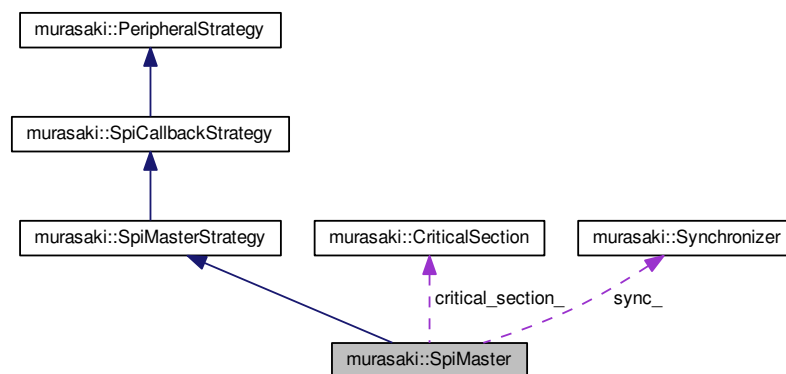
Thread safe, synchronous and blocking IO. Encapsulating SPI master. Based on STM32Cube HAL driver and FreeRTOS.

```
#include <spimaster.hpp>
```

Inheritance diagram for murasaki::SpiMaster:



Collaboration diagram for `murasaki::SpiMaster`:



## Public Member Functions

- [SpiMaster](#) (`SPI_HandleTypeDef *spi_handle`)

*Constructor.*

- virtual [SpiStatus TransmitAndReceive](#) (`murasaki::SpiSlaveAdapterStrategy *spi_spec`, `const uint8_t *tx_data`, `uint8_t *rx_data`, `unsigned int size`, `unsigned int timeout_ms=murasaki::kwmsIndefinitely`)

*Data transfer to/from SPI slave.*

- virtual `bool` [TransmitAndReceiveCompleteCallback](#) (`void *ptr`)

*Callback to notify the end of transfer.*

- virtual `bool` [HandleError](#) (`void *ptr`)

*Error handling.*

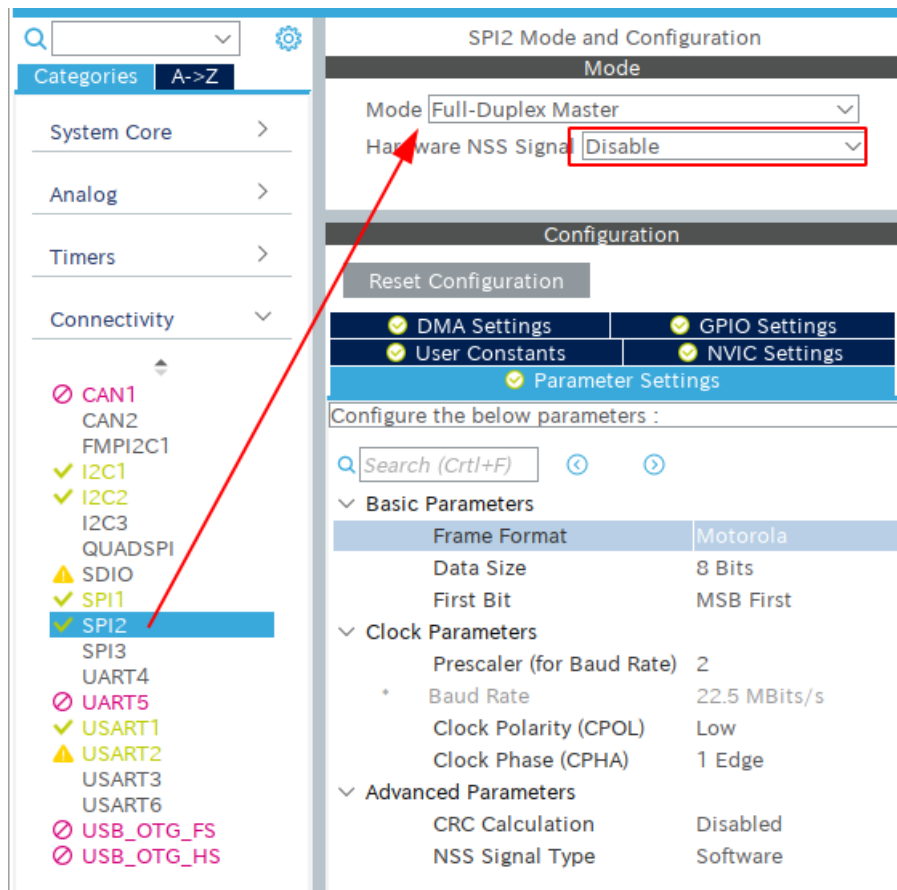
## Additional Inherited Members

### 7.36.1 Detailed Description

The [SpiMaster](#) class is the wrapper of the SPI controller.

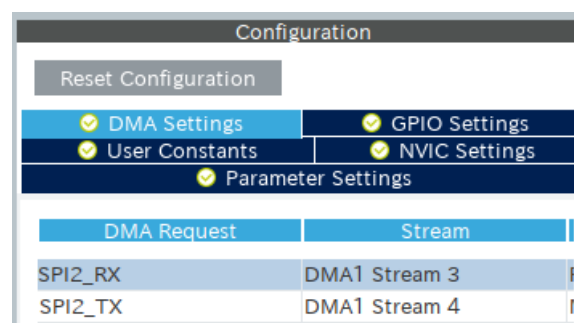
## Configuration

To configure the SPI peripheral as master, chose SPI peripheral in the Device Configuration Tool of the CubeIDE. Set it as SPI Duplex Master

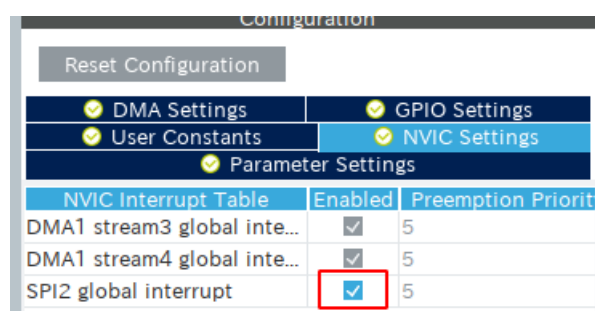


[SpiMaster](#) class supports the Duplex master, 8bit mode only. The clock phase and Clock polarity is not the matter at the configuration phase.

The DMA have to be enabled for both TX and RX. The data size is 8bit for both Peripheral and memory.



And then, enable the interrupt.



### Creating a peripheral object

To use the `SpiMaster` class, create an instance with `SPI_HandleTypeDef * type` pointer. For example, to create an instance for the SPI3 peripheral :

```
my_spi3 = new murasaki::SpiMaster(&hspi3);
```

Where `hspi3` is the handle generated by CubeIDE for SPI3 peripheral.

### Handling an interrupt

Interrupt is handled automatically. Programmer doesn't need to care.

### Transmitting and Receiving

Once the instance and callback are correctly prepared, we can use the `Transfer` member function.

The `SpiMaster::TransmitAndReceive()` member function is an asynchronous function. A programmer can specify the timeout by `timeout_ms` parameter. By default, this parameter is set by `kwmsIndefinitely` which specifies never time out.

Both member functions can be called from only the task context. If these are called in the ISR context, the result is unknown.

Note : The behavior of when the timeout happen is not tested. Actually, it should not happen because DMA is taken in SPI transmission. Murasaki stops internal DMA, interrupt and SPI processing internally then, return.

Other error will cause the re-initializing of the SPI master. Murasaki doesn't support any of CRC detection, TI frame mode or Multi-master SPI.

## 7.36.2 Constructor & Destructor Documentation

### 7.36.2.1 murasaki::SpiMaster::SpiMaster ( SPI\_HandleTypeDef \* spi\_handle )

#### Parameters

<i>spi_handle</i>	Handle to the SPI peripheral. This have to be configured to use DMA by CubeIDE.
-------------------	---

## 7.36.3 Member Function Documentation

### 7.36.3.1 bool murasaki::SpiMaster::HandleError ( void \* ptr ) [virtual]

#### Parameters

<i>ptr</i>	Pointer to <code>I2C_HandleTypeDef</code> struct.
------------	---

**Returns**

true: ptr matches with device and handle the error. false : doesn't match.

A handle to print out the error message.

Checks whether handle has error and if there is, print appropriate error. Then return.

This function have to be called from HAL\_SPI\_ErrorCallback.

```
void HAL_SPI_ErrorCallback(SPI_HandleTypeDef *hspi) {
    if ( murasaki::platform.spil->HandleError(hspi) )
        return;
}
```

Implements [murasaki::SpiCallbackStrategy](#).

**7.36.3.2 SpiStatus** `murasaki::SpiMaster::TransmitAndReceive ( murasaki::SpiSlaveAdapterStrategy * spi_spec, const uint8_t * tx_data, uint8_t * rx_data, unsigned int size, unsigned int timeout_ms = murasaki::kwmsIndefinitely ) [virtual]`

**Parameters**

<i>spi_spec</i>	A pointer to the AbstractSpiSpecification to specify the slave device.
<i>tx_data</i>	Data to be transmitted
<i>rx_data</i>	Data buffer to receive data
<i>size</i>	Transfer data size [byte] for each way.
<i>timeout_ms</i>	Timeout limit [mS]

**Returns**

true if transfer complete, false if timeout

Transfer the data to/from SPI slave specified by parameter spi\_spec.

This member function re-initialize the SPI peripheral based on the clock information from the spi\_spec. And then, assert the chips elect through the spi\_spec during the data transfer.

Following are the return codes:

- [murasaki::kspisOK](#) : The transfer complete without error.
- [murasaki::kspisModeCRC](#) : CRC error was detected.
- [murasaki::kspisOverflow](#) : SPI overflow or underflow was detected.
- [murasaki::kspisFrameError](#) Frame error in TI mode.
- [murasaki::kspisDMA](#) : Some DMA error was detected in HAL. SPI re-initialized.
- [murasaki::kspisErrorFlag](#) : Unhandled flags. SPI re-initialized.
- [murasaki::ki2csTimeOut](#) : Timeout detected. DMA stopped.
- Other : Unhandled error . SPI re-initialized.

Implements [murasaki::SpiMasterStrategy](#).

**7.36.3.3** `bool murasaki::SpiMaster::TransmitAndReceiveCompleteCallback ( void * ptr ) [virtual]`

**Parameters**

<i>ptr</i>	Pointer to the control object.
------------	--------------------------------

**Returns**

true: ptr matches with device and handle the error. false : doesn't match.

This function have to be called from HAL\_SPI\_TxRxCpltCallback

```
void HAL_SPI_TxRxCpltCallback(SPI_HandleTypeDef *hspi) {
    if ( murasaki::platform.spi1->TransmitAndReceiveCompleteCallback(hspi) )
        return;
}
```

Implements [murasaki::SpiCallbackStrategy](#).

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

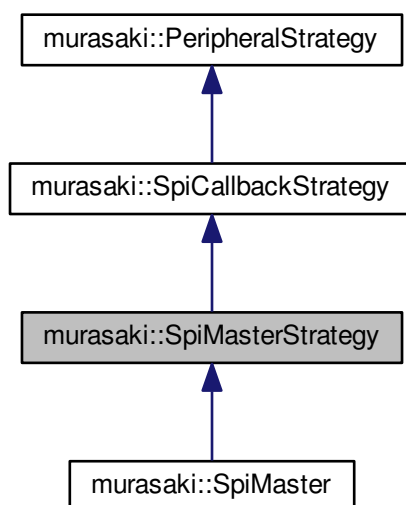
- [spimaster.hpp](#)
- [spimaster.cpp](#)

**7.37 murasaki::SpiMasterStrategy Class Reference**

Root class of the SPI master.

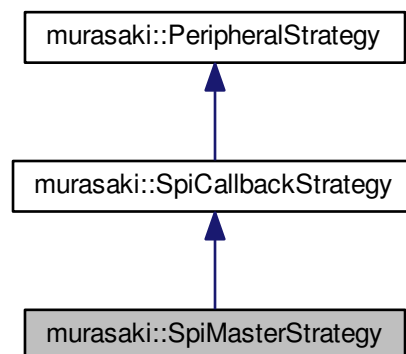
```
#include <spimasterstrategy.hpp>
```

Inheritance diagram for murasaki::SpiMasterStrategy:





Collaboration diagram for murasaki::SpiMasterStrategy:



#### Public Member Functions

- virtual [SpiStatus TransmitAndReceive](#) ([murasaki::SpiSlaveAdapterStrategy](#) \*spi\_spec, const uint8\_t \*tx\_data, uint8\_t \*rx\_data, unsigned int size, unsigned int timeout\_ms=[murasaki::kwmsIndefinitely](#))=0  
*Thread safe, synchronous SPI transfer.*

#### Additional Inherited Members

##### 7.37.1 Detailed Description

This class provides a thread safe, synchronous SPI transfer.

##### 7.37.2 Member Function Documentation

**7.37.2.1** virtual `SpiStatus` `murasaki::SpiMasterStrategy::TransmitAndReceive` ( `murasaki::SpiSlaveAdapterStrategy` \* *spi\_spec*, const uint8\_t \* *tx\_data*, uint8\_t \* *rx\_data*, unsigned int *size*, unsigned int *timeout\_ms* = `murasaki::kwmsIndefinitely` ) [pure virtual]

#### Parameters

<i>spi_spec</i>	Pointer to the SPI slave adapter which has clock configuraiton and chip select handling.
<i>tx_data</i>	Data to be transmitted
<i>rx_data</i>	Data buffer to receive data
<i>size</i>	Transfer data size [byte] for each way. Must be smaller than 65536
<i>timeout_ms</i>	Timeout limit [mS]

**Returns**

true if transfer complete, false if timeout

Implemented in [murasaki::SpiMaster](#).

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

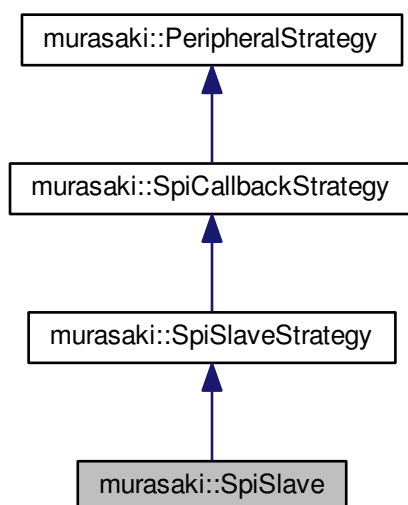
- [spimasterstrategy.hpp](#)

### 7.38 murasaki::SpiSlave Class Reference

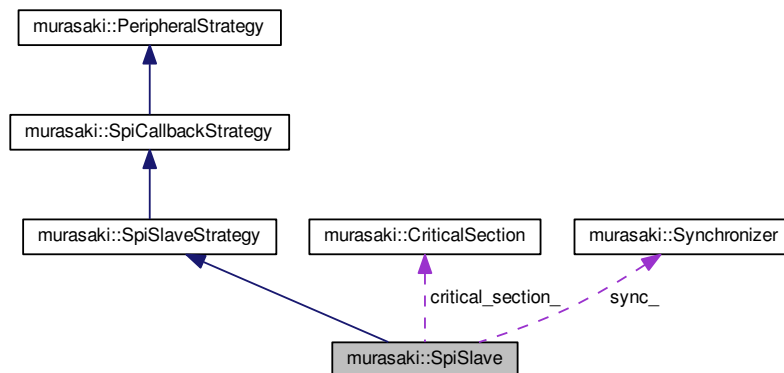
Thread safe, synchronous and blocking IO. Encapsulating SPI slave. Based on STM32Cube HAL driver and FreeRTOS.

```
#include <spislave.hpp>
```

Inheritance diagram for murasaki::SpiSlave:



Collaboration diagram for murasaki::SpiSlave:



#### Public Member Functions

- [SpiSlave](#) (SPI\_HandleTypeDef \*spi\_handle)  
*Constructor.*
- virtual [SpiStatus TransmitAndReceive](#) (const uint8\_t \*tx\_data, uint8\_t \*rx\_data, unsigned int size, unsigned int \*transferred\_count, unsigned int timeout\_ms=[murasaki::kwmsIndefinitely](#))  
*Data transfer to/from SPI slave.*
- virtual bool [TransmitAndReceiveCompleteCallback](#) (void \*ptr)  
*Callback to notify the end of transfer.*
- virtual bool [HandleError](#) (void \*ptr)  
*Error handling.*

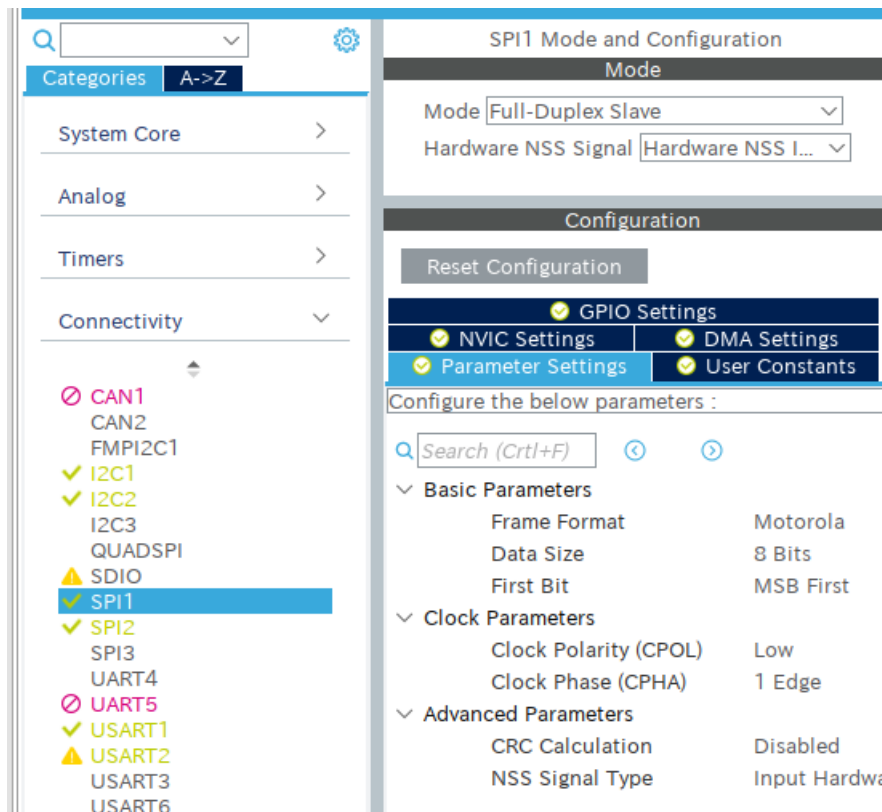
#### Additional Inherited Members

##### 7.38.1 Detailed Description

The [SpiSlave](#) class is the wrapper of the SPI controller.

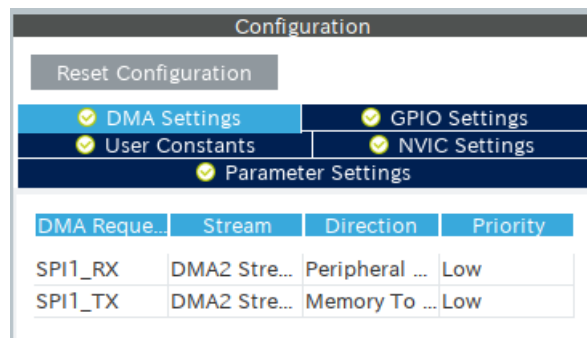
#### Configuration

To configure the SPI peripheral as slave, chose SPI peripheral in the Device Configuration Tool of the CubeIDE. Set it as SPI Duplex Slave

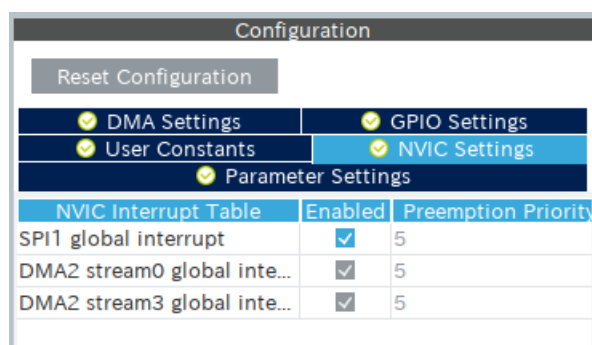


[SpiMaster](#) class supports the Duplex slave, 8bit mode only. Also the clock phase and Clock polarity have to be configured by the configuration tool. This is different from master.

The DMA have to be enabled for both TX and RX. The data size is 8bit for both Peripheral and memory.



And then, enable the interrupt.



### Creating a peripheral object

To use the [SpiSlave](#) class, make an instance with `SPI_HandleTypeDef *` type pointer. For example, to create an instance for the SPI3 peripheral :

```
my_spi3 = new murasaki::SpiSlave(&hspi3);
```

Where `hspi3` is the handle generated by CubeIDE for SPI3 peripheral.

### Handling an interrupt

Interrupt is handled automatically. Programmer doesn't need to care.

### Transmitting and Receiving

Once the instance and callback are correctly prepared, we can use the `Transfer` member function.

The [SpiSlave::TransmitAndReceive\(\)](#) member function is a synchronous function. A programmer can specify the timeout by `timeout_ms` parameter. By default, this parameter is set by `kwmsIndefinitely` which specifies never time out.

This member function can be called from only the task context. If these are called in the ISR context, the result is unknown.

Other error will cause the re-initializing of the SPI slave. Murasaki doesn't support any of CRC detection, TI frame mode or Multi-master SPI.

## 7.38.2 Constructor & Destructor Documentation

### 7.38.2.1 murasaki::SpiSlave::SpiSlave ( SPI\_HandleTypeDef \* *spi\_handle* )

#### Parameters

<i>spi_handle</i>	Handle to the SPI peripheral. This have to be configured to use DMA by CubeIDE.
-------------------	---

## 7.38.3 Member Function Documentation

### 7.38.3.1 bool murasaki::SpiSlave::HandleError ( void \* *ptr* ) [virtual]

#### Parameters

<i>ptr</i>	Pointer to <code>I2C_HandleTypeDef</code> struct.
------------	---

#### Returns

true: *ptr* matches with device and handle the error. false : doesn't match.

A handle to print out the error message.

Checks whether handle has error and if there is, print appropriate error. Then return.

This member function have to be called from [HAL\\_SPI\\_ErrorCallback\(\)](#)

```
void HAL_SPI_ErrorCallback(SPI_HandleTypeDef * hspi) {
    if ( murasaki::platform.spil->HandleError(hspi) )
        return;
}
```

Implements [murasaki::SpiCallbackStrategy](#).

**7.38.3.2 SpiStatus** [murasaki::SpiSlave::TransmitAndReceive](#) ( const uint8\_t \* *tx\_data*, uint8\_t \* *rx\_data*, unsigned int *size*, unsigned int \* *transferred\_count*, unsigned int *timeout\_ms* = [murasaki::kwmsIndefinitely](#) ) [virtual]

#### Parameters

<i>tx_data</i>	Data to be transmitted
<i>rx_data</i>	Data buffer to receive data
<i>size</i>	Transfer data size [byte] for each way.
<i>transferred_count</i>	( Currently, Just ignored) The transfered number of bytes during API.
<i>timeout_ms</i>	Timeout limit [mS]

#### Returns

true if transfer complete, false if timeout

Transfer the data to/from SPI slave specified by parameter *spi\_spec*.

This member function re-initialize the SPI peripheral based on the clock information from the *spi\_spec*. And then, assert the chips elect through the *spi\_spec* during the data transfer.

Following are the return codes:

- [murasaki::kspisOK](#) : The transfer complete without error.
- [murasaki::kspisModeCRC](#) : CRC error was detected.
- [murasaki::kspisOverflow](#) : SPI overflow or underflow was detected.
- [murasaki::kspisFrameError](#) Frame error in TI mode.
- [murasaki::kspisDMA](#) : Some DMA error was detected in HAL. SPI re-initialized.
- [murasaki::kspisErrorFlag](#) : Unhandled flags. SPI re-initialized.
- [murasaki::ki2csTimeOut](#) : Timeout detected. DMA stopped.
- Other : Unhandled error . SPI re-initialized.

Implements [murasaki::SpiSlaveStrategy](#).

**7.38.3.3** [bool](#) [murasaki::SpiSlave::TransmitAndReceiveCompleteCallback](#) ( void \* *ptr* ) [virtual]

#### Parameters

<i>ptr</i>	Pointer to the control object.
------------	--------------------------------

#### Returns

true if no error.

This member function have to be called from [HAL\\_SPI\\_TxRxCpltCallback\(\)](#)

```
void HAL_SPI_TxRxCpltCallback (SPI_HandleTypeDef * hspi)
{
    if ( murasaki::platform.spil->TransmitAndReceiveCompleteCallback(hspi))
        return;
}
```

Implements [murasaki::SpiCallbackStrategy](#).

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

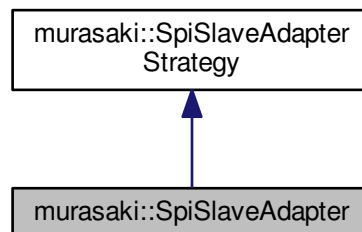
- [spislave.hpp](#)
- [spislave.cpp](#)

## 7.39 murasaki::SpiSlaveAdapter Class Reference

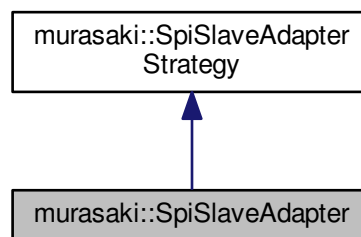
A speticier of SPI slave.

```
#include <spislaveadapter.hpp>
```

Inheritance diagram for murasaki::SpiSlaveAdapter:



Collaboration diagram for `murasaki::SpiSlaveAdapter`:



### Public Member Functions

- [SpiSlaveAdapter](#) ([murasaki::SpiClockPolarity](#) pol, [murasaki::SpiClockPhase](#) pha, ::GPIO\_TypeDef \*port, uint16\_t pin)  
*Constructor.*
- [SpiSlaveAdapter](#) (unsigned int pol, unsigned int pha, ::GPIO\_TypeDef \*const port, uint16\_t pin)  
*Constructor.*
- virtual void [AssertCs](#) ()  
*Chip select assertion.*
- virtual void [DeassertCs](#) ()  
*Chip select deassertoin.*

### Additional Inherited Members

#### 7.39.1 Detailed Description

This class describes how this slave is. The description is clock POL and PHA for the specific slave device.

In addition to the clock porality, the instans of this class works as salogate of the chip select control.

The instans will be passed to the [SpiMaster](#) class.

#### 7.39.2 Constructor & Destructor Documentation

7.39.2.1 `murasaki::SpiSlaveAdapter::SpiSlaveAdapter ( murasaki::SpiClockPolarity pol, murasaki::SpiClockPhase pha, ::GPIO_TypeDef * port, uint16_t pin )`

#### Parameters

<i>pol</i>	Polarity setting
<i>pha</i>	Phase setting
<i>port</i>	GPIO port of the chip select
<i>pin</i>	GPIO pin of the chip select



The port and pin parameters are passed to the HAL\_GPIO\_WritePin(). The port and pin have to be configured by CubeIDE correctly.

7.39.2.2 murasaki::SpiSlaveAdapter::SpiSlaveAdapter ( unsigned int *pol*, unsigned int *pha*, ::GPIO\_TypeDef \*const *port*, uint16\_t *pin* )

#### Parameters

<i>pol</i>	Polarity setting
<i>pha</i>	Phase setting
<i>port</i>	GPIO port of the chip select
<i>pin</i>	GPIO pin of the chip select

The port and pin parameters are passed to the HAL\_GPIO\_WritePin(). The port and pin have to be configured by CubeIDE correctly.

#### 7.39.3 Member Function Documentation

7.39.3.1 void murasaki::SpiSlaveAdapter::AssertCs ( ) [virtual]

This member function asset the output line to select the slave chip.

Reimplemented from [murasaki::SpiSlaveAdapterStrategy](#).

7.39.3.2 void murasaki::SpiSlaveAdapter::DeassertCs ( ) [virtual]

This member function deasset the output line to de-select the slave chip.

Reimplemented from [murasaki::SpiSlaveAdapterStrategy](#).

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

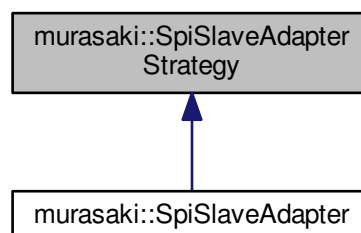
- [spislaveadapter.hpp](#)
- [spislaveadapter.cpp](#)

## 7.40 murasaki::SpiSlaveAdapterStrategy Class Reference

Definition of the root class of SPI slave adapter.

```
#include <spislaveadapterstrategy.hpp>
```

Inheritance diagram for murasaki::SpiSlaveAdapterStrategy:



## Public Member Functions

- [SpiSlaveAdapterStrategy](#) ([murasaki::SpiClockPolarity](#) pol, [murasaki::SpiClockPhase](#) pha)  
*Constructor.*
- [SpiSlaveAdapterStrategy](#) (unsigned int pol, unsigned int pha)  
*Constructor.*
- virtual [~SpiSlaveAdapterStrategy](#) ()  
*Destructor.*
- virtual void [AssertCs](#) ()  
*Chip select assertion.*
- virtual void [DeassertCs](#) ()  
*Chip select deassertoin.*
- [murasaki::SpiClockPhase](#) [GetCpha](#) ()  
*Getter of the CPHA.*
- [murasaki::SpiClockPolarity](#) [GetCpol](#) ()  
*Getter of the CPOL.*

## Protected Attributes

- [murasaki::SpiClockPolarity](#) const [cpol\\_](#)  
*Setting of CPOL.*
- [murasaki::SpiClockPhase](#) const [cpha\\_](#)  
*Setting of CPHA.*

### 7.40.1 Detailed Description

A prototype of the SPI slave device adapter.

The adapter adds the following SPI attributes :

- CPOL
- CPHA
- Chip select control for slave.

Because SPI slave has different setting device by device, this adapter should be passed to the each transactions.

[AssetCs\(\)](#) and [DeassertCs\(\)](#) have to be overridden to control the chip select output. These member functions will be called from the [AbstractSpiMaster](#).

### 7.40.2 Constructor & Destructor Documentation

#### 7.40.2.1 [murasaki::SpiSlaveAdapterStrategy::SpiSlaveAdapterStrategy](#) ( [murasaki::SpiClockPolarity](#) pol, [murasaki::SpiClockPhase](#) pha )

#### Parameters

<i>pol</i>	Polarity setting
<i>pha</i>	Phase setting

7.40.2.2 murasaki::SpiSlaveAdapterStrategy::SpiSlaveAdapterStrategy ( unsigned int *pol*, unsigned int *pha* )

## Parameters

<i>pol</i>	Polarity setting
<i>pha</i>	Phase setting

## 7.40.3 Member Function Documentation

## 7.40.3.1 void murasaki::SpiSlaveAdapterStrategy::AssertCs ( ) [virtual]

This member function asset the output line to select the slave chip.

This have to be overridden.

Reimplemented in [murasaki::SpiSlaveAdapter](#).

## 7.40.3.2 void murasaki::SpiSlaveAdapterStrategy::DeassertCs ( ) [virtual]

This member function deasset the output line to de-select the slave chip.

This have to be overridden.

Reimplemented in [murasaki::SpiSlaveAdapter](#).

## 7.40.3.3 murasaki::SpiClockPhase murasaki::SpiSlaveAdapterStrategy::GetCpha ( )

## Returns

CPHA setting

## 7.40.3.4 murasaki::SpiClockPolarity murasaki::SpiSlaveAdapterStrategy::GetCpol ( )

## Returns

CPOL setting

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

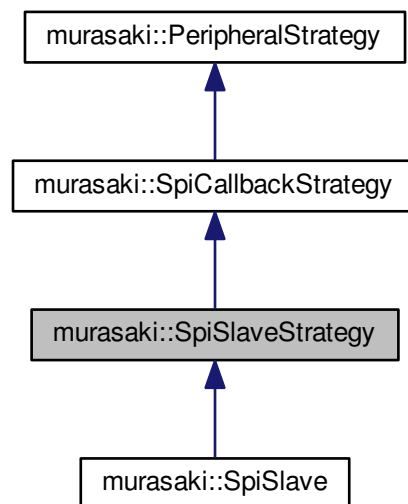
- [spislaveadapterstrategy.hpp](#)
- [spislaveadapterstrategy.cpp](#)

## 7.41 murasaki::SpiSlaveStrategy Class Reference

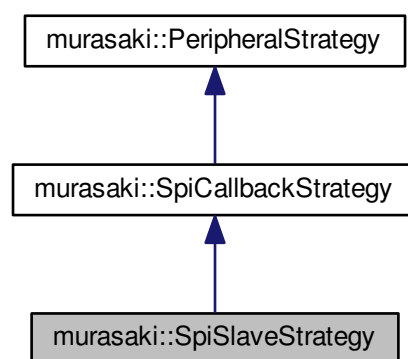
Root class of the SPI slave.

```
#include <spislavestrategy.hpp>
```

Inheritance diagram for murasaki::SpiSlaveStrategy:



Collaboration diagram for murasaki::SpiSlaveStrategy:



### Public Member Functions

- virtual [SpiStatus TransmitAndReceive](#) (const uint8\_t \*tx\_data, uint8\_t \*rx\_data, unsigned int size, unsigned int \*transferred\_count=nullptr, unsigned int timeout\_ms=[murasaki::kwmsIndefinitely](#))=0  
*Thread safe, synchronous SPI transfer.*

## Additional Inherited Members

## 7.41.1 Detailed Description

This class provides a thread safe, synchronous SPI transfer.

## 7.41.2 Member Function Documentation

7.41.2.1 virtual **SpiStatus** murasaki::SpiSlaveStrategy::TransmitAndReceive ( const uint8\_t \* *tx\_data*, uint8\_t \* *rx\_data*, unsigned int *size*, unsigned int \* *transferred\_count* = nullptr, unsigned int *timeout\_ms* = **murasaki::kwmsIndefinitely** ) [pure virtual]

## Parameters

<i>tx_data</i>	Data to be transmitted
<i>rx_data</i>	Data buffer to receive data
<i>size</i>	Transfer data size [byte] for each way. Must be smaller than 65536
<i>transferred_count</i>	The transferred number of bytes during API.
<i>timeout_ms</i>	Timeout limit [mS]

## Returns

true if transfer complete, false if timeout

Implemented in [murasaki::SpiSlave](#).

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

- [spislavestrategy.hpp](#)

## 7.42 murasaki::Synchronizer Class Reference

Synchronization class between a task and interrupt. This class provide the synchronization between a task and interrupt.

```
#include <synchronizer.hpp>
```

## Public Member Functions

- [Synchronizer](#) ()  
*Constructor. Assigning semaphore internally.*
- virtual [~Synchronizer](#) ()  
*Destructor. Deleting semaphore internally.*
- bool [Wait](#) (unsigned int timeout\_ms=[kwmsIndefinitely](#))  
*Let the task wait for an interrupt.*
- void [Release](#) ()  
*Release the task.*

### 7.42.1 Detailed Description

Synchronization mean, task waits for a interrupt by calling `InterruptSynchronizer::WaitForInterruptFromTask()` and during the wait, task yields the cpu to other task. So, CPU can do other job during a task is waiting for interrupt. Interrupt will allow task run again by `InterruptSynchronizer::ReleasetaskFromISR()` member function.

### 7.42.2 Member Function Documentation

#### 7.42.2.1 `void murasaki::Synchronizer::Release ( )`

Release the task waiting. This member function can be called from both task and the interrupt context.

#### 7.42.2.2 `bool murasaki::Synchronizer::Wait ( unsigned int timeout_ms = kwmsIndefinitely )`

##### Parameters

<code><i>timeout_ms</i></code>	Timeout by millisecond. The default value let the task wait for interrupt forever.
--------------------------------	--

##### Returns

True if interrupt came before timeout. False if timeout happen.

This member function have to be called from the task context. Otherwise, the behavior is not predictable.

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

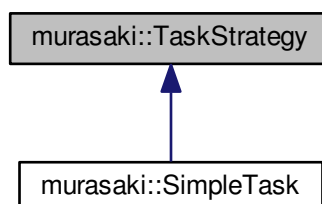
- [synchronizer.hpp](#)
- [synchronizer.cpp](#)

## 7.43 murasaki::TaskStrategy Class Reference

A mother of all tasks.

```
#include <taskstrategy.hpp>
```

Inheritance diagram for `murasaki::TaskStrategy`:



## Public Member Functions

- [TaskStrategy](#) (const char \*task\_name, unsigned short stack\_depth, [murasaki::TaskPriority](#) task\_priority, const void \*task\_parameter)  
*Contractor. Task entity is not created here.*
- virtual [~TaskStrategy](#) ()  
*Destructor.*
- void [Start](#) ()  
*Create a task and run it.*
- void [Resume](#) ()  
*Resume a task from suspended state.*
- void [Suspend](#) ()  
*Put the task into suspend state.*
- const char \* [GetName](#) ()  
*Get a name of task.*
- unsigned int [getStackDepth](#) ()  
*Obtain the size of the stack.*
- int [getStackMinHeadroom](#) ()  
*Obtain the headroom of the stack.*

## Protected Member Functions

- virtual void [TaskBody](#) (const void \*ptr)=0  
*Actual task entity. Must be overridden by programmer.*

## Static Protected Member Functions

- static void [Launch](#) (void \*ptr)  
*Internal use only. Create a task from [TaskBody\(\)](#)*

## 7.43.1 Detailed Description

Encapsulate a FreeRTOS task.

The constructor just stores given parameter internally. And then, these parameter is passed to a task when [Start\(\)](#) member function is called. Actual task creation is done inside [Start\(\)](#).

The destructor deletes the task. Releasing task from all the resources ( ex: semaphore ) before deleting, is the responsibility of the programmer.

Base on the description at [http://idken.net/posts/2017-02-01-freertos\\_task\\_cpp/](http://idken.net/posts/2017-02-01-freertos_task_cpp/)

## 7.43.2 Constructor &amp; Destructor Documentation

- 7.43.2.1 `murasaki::TaskStrategy::TaskStrategy ( const char * task_name, unsigned short stack_depth, murasaki::TaskPriority task_priority, const void * task_parameter )`

## Parameters

<i>task_name</i>	Name of task. Will be passed to task when started.
<i>stack_depth</i>	[Byte]
<i>task_priority</i>	Priority of the task. from 1 to up to configMAX_PRIORITIES -1. The high number is the high priority.
<i>task_parameter</i>	Optional parameter to the task.

## 7.43.3 Member Function Documentation

7.43.3.1 `const char * murasaki::TaskStrategy::GetName ( )`

## Returns

A name of task.

7.43.3.2 `unsigned int murasaki::TaskStrategy::getStackDepth ( )`

## Returns

Total depth of the task stack [byte]

7.43.3.3 `int murasaki::TaskStrategy::getStackMinHeadroom ( )`

## Returns

The remained headroom in stack [byte]. 0 mean stack is overflown. -1 mean Stack overflow check is not provided.

Return value is the avairable stack size in byte.

Internally, this function uses [Stack Usage and Stack Overflow Checking](#).

Thus,

- INCLUDE\_uxTaskGetStackHighWaterMark have to be non zero
- configCHECK\_FOR\_STACK\_OVERFLOW have to be non zero

If above conditions are not met, this function returns -1.

7.43.3.4 `void murasaki::TaskStrategy::Launch ( void * ptr ) [static], [protected]`

## Parameters

<i>ptr</i>	passing "this" pointer.
------------	-------------------------

7.43.3.5 `void murasaki::TaskStrategy::Start ( void )`

A task is created with given parameter to the constructors and then run.



7.43.3.6 `virtual void murasaki::TaskStrategy::TaskBody ( const void * ptr )` `[protected],[pure virtual]`

#### Parameters

<i>ptr</i>	Optional parameter to the task body. This ptr is copied from the task_parameter of the Constructor.
------------	---

The task body is called only once as task entity. Programmer have to override this member function with his/her own [TaskBody\(\)](#).

From this member function, class members are able to access.

Implemented in [murasaki::SimpleTask](#).

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

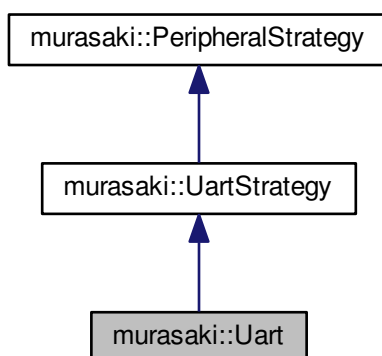
- [taskstrategy.hpp](#)
- [taskstrategy.cpp](#)

## 7.44 murasaki::Uart Class Reference

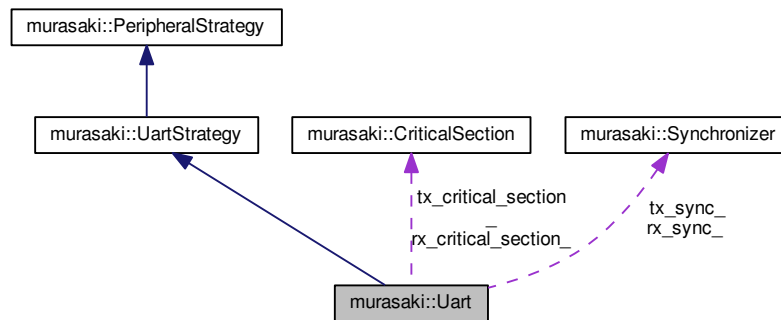
Thread safe, synchronous and blocking IO. Concrete implementation of UART controller. Based on the STM32Cube HAL DMA Transfer.

```
#include <uart.hpp>
```

Inheritance diagram for murasaki::Uart:



Collaboration diagram for `murasaki::Uart`:



### Public Member Functions

- **Uart** (UART\_HandleTypeDef \*uart)  
*Constructor.*
- virtual **~Uart** ()  
*Destructor. Delete internal variables.*
- virtual void **SetHardwareFlowControl** (UartHardwareFlowControl control)  
*Set the behavior of the hardware flow control.*
- virtual void **SetSpeed** (unsigned int baud\_rate)  
*Set the BAUD rate.*
- virtual **murasaki::UartStatus Transmit** (const uint8\_t \*data, unsigned int size, unsigned int timeout\_ms)  
*Transmit raw data through an UART by synchronous mode.*
- virtual **murasaki::UartStatus Receive** (uint8\_t \*data, unsigned int count, unsigned int \*transferred\_count, **UartTimeout** uart\_timeout, unsigned int timeout\_ms)  
*Receive raw data through an UART by synchronous mode.*
- virtual bool **TransmitCompleteCallback** (void \*const ptr)  
*Call back for entire block transfer completion.*
- virtual bool **ReceiveCompleteCallback** (void \*const ptr)  
*Call back for entire block transfer completion.*
- virtual bool **HandleError** (void \*const ptr)  
*Error handling.*

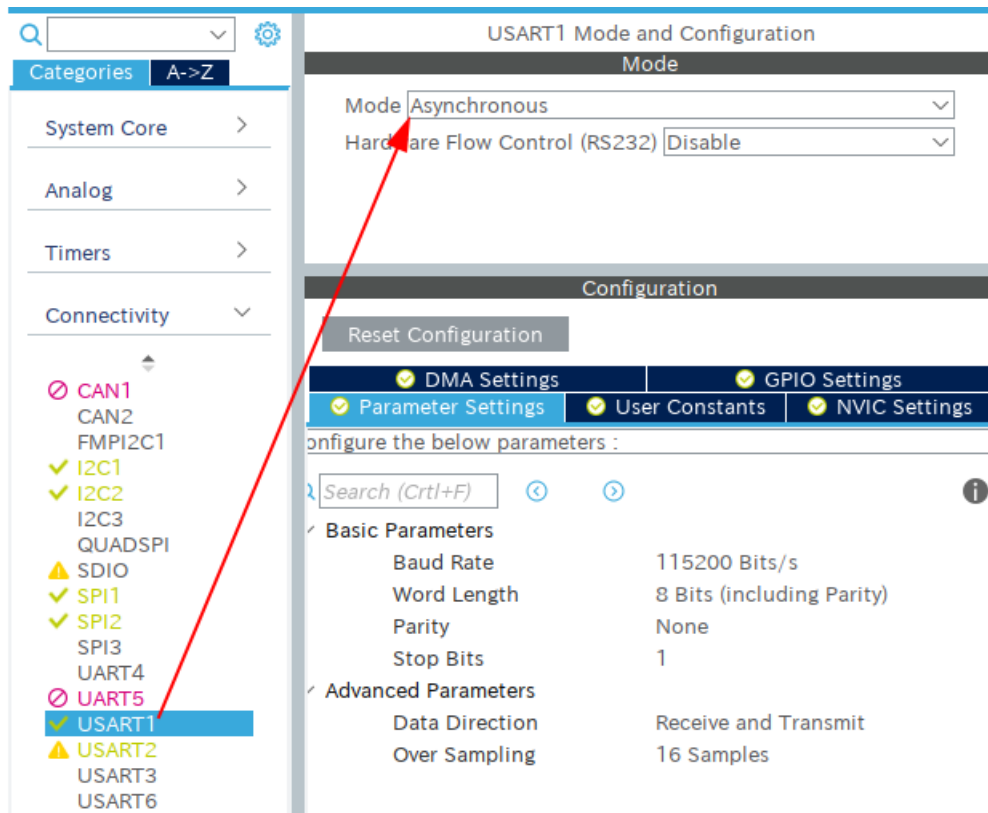
### Additional Inherited Members

#### 7.44.1 Detailed Description

The **Uart** class is a wrapper of the UART controller.

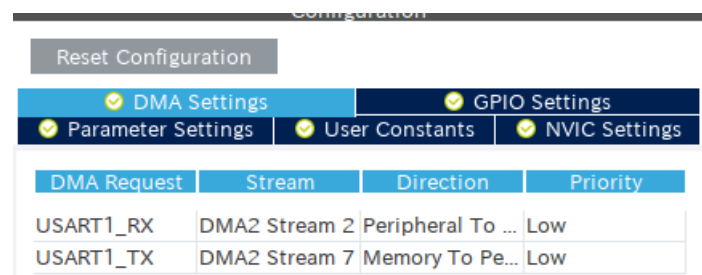
## Configuration

To configure the UART peripheral, chose UART/USART peripheral in the Device Configuration Tool of the CubeIDE. Set it as Asynchronous mode.

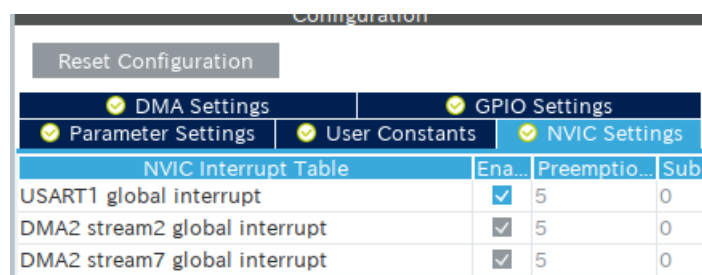


Make sure setting direction to Receive and Transmit. Other parameters are up to the application.

The DMA have to be enabled for both TX and RX. The data size is 8bit for both Peripheral and memory.



And then, enable the interrupt.



## Creating a peripheral object

To use the `Uart` class, create an instance with `UART_HandleTypeDef * type` pointer. For example, to create an instance for the UART3 peripheral :

```
my_uart3 = new murasaki::Uart (&huart3);
```

Where `huart3` is the handle generated by CubeIDE for UART3 peripheral. To use this class, the UART peripheral have to be configured to use the DMA functionality. The baud rate, length and flow control should be configured by the CubeIDE.

## Handling an interrupt

Interrupt is handled automatically. Programmer doesn't need to care.

## Transmitting and Receiving

Once the instance and callbacks are correctly prepared, we can use the Tx/Rx member function.

The `Uart::Transmit()` member function is a synchronous function. A programmer can specify the timeout by `timeout_ms` parameter. By default, this parameter is set by `murasaki::kwmsIndefinitely` which specifies never time out.

The `Uart::Receive()` member function is a synchronous function. A programmer can specify the timeout by `timeout_ms` parameter. By default, this parameter is set by `murasaki::kwmsIndefinitely` which specifies never time out.

Both methods can be called from only the task context. If these are called in the ISR context, the result is unknown.

## 7.44.2 Constructor & Destructor Documentation

### 7.44.2.1 murasaki::Uart::Uart ( UART\_HandleTypeDef \* uart )

#### Parameters

<i>uart</i>	Pointer to a UART control struct. This device have to be configured to use DMA and interrupt for both Tx and Rx.
-------------	--

Store the given `uart` pointer into the internal variable. This pointer is passed to the STM32Cube HAL UART functions when needed.

## 7.44.3 Member Function Documentation

### 7.44.3.1 bool murasaki::Uart::HandleError ( void \*const ptr ) [virtual]

#### Parameters

<i>ptr</i>	Pointer to <code>UART_HandleTypeDef</code> struct.
------------	--

**Returns**

true: ptr matches with UART device and handle the error. false : doesn't match.

A handle to print out the error message.

Checks whether handle has error and if there is, print appropriate error. Then return.

This function have to be coalled from().

```
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart) {
    if (murasaki::platform.uart_console->HandleError(huart))
        return;
}
```

Implements [murasaki::UartStrategy](#).

**7.44.3.2 murasaki::UartStatus** [murasaki::Uart::Receive](#) ( uint8\_t \* *data*, unsigned int *count*, unsigned int \* *transfered\_count*, [UartTimeout](#) *uart\_timeout*, unsigned int *timeout\_ms* ) [virtual]

**Parameters**

<i>data</i>	Data buffer to place the received data..
<i>count</i>	The count of the data ( byte ) to be transfered. Must be smaller than 65536
<i>transfered_count</i>	( Currently, Just ignored) Number of bytes transfered. The nullPtr means no need to return value.
<i>uart_timeout</i>	Specify <a href="#">murasaki::kutIdleTimeout</a> , if idle line timeout is needed.
<i>timeout_ms</i>	Time out limit by milliseconds.

**Returns**

True if all data transfered completely. False if time out happen.

Receive to given data buffer through an UART device.

The receiving mode is synchronous. That means, function returns when specified number of data has been received, except timeout. Passing [murasaki::kwmsIndefinitely](#) to the parameter *timeout\_ms* orders not to return until complete receiving. Other value of *timeout\_ms* parameter specifies the time out by millisecond. If time out happen, function returns false. If not happen, it returns true.

This function is exclusive. Internally this function is guarded by mutex. Then this function is thread safe. This function is forbidden to call from ISR.

The return values are:

- [murasaki::kursOK](#) : Transmit complete.
- [murasaki::kursTimeOut](#) : Time out occur.
- [murasaki::kursOverrun](#) : Next char was written to TX register. This is fatal problem in HAL. Peripheral is re-initialized internally.
- [murasaki::kursDMA](#) : This is fatal problem in HAL. Peripheral is re-initialized internally.
- other : This is fatal problem in HAL. Peripheral is re-initialized internally.

Implements [murasaki::UartStrategy](#).

**7.44.3.3 bool** [murasaki::Uart::ReceiveCompleteCallback](#) ( void \*const *ptr* ) [virtual]

**Parameters**

<i>ptr</i>	Pointer to UART_HandleTypeDef struct.
------------	---------------------------------------

**Returns**

true: ptr matches with UART device and handle the call back. false : doesn't match.

A call back to notify the end of entire block transfer. This is considered as the end of DMA based receiving. The context have to be interrupt.

This member function checks whether the given ptr parameter matches its own device, and if matched, Release the waiting task and return true. If it doesn't match, just return false.

This method have to be called from [HAL\\_UART\\_RxCpltCallback\(\)](#). See STM32F7 HAL manual for detail

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (murasaki::platform.uart_console->ReceiveCompleteCallback(huart))
        return;
}
```

Implements [murasaki::UartStrategy](#).

**7.44.3.4** void [murasaki::Uart::SetHardwareFlowControl](#) ( [UartHardwareFlowControl control](#) ) [virtual]

**Parameters**

<i>control</i>	The control mode.
----------------	-------------------

Before calling this method, all transmission and receive activities have to be finished. This is responsibility of the programmer.

Note this method is NOT re-entrant. In other word, this member function can be called from both task and interrupt context.

Reimplemented from [murasaki::UartStrategy](#).

**7.44.3.5** void [murasaki::Uart::SetSpeed](#) ( unsigned int *baud\_rate* ) [virtual]

**Parameters**

<i>baud_rate</i>	BAUD rate ( 110, 300,... 57600,... )
------------------	--------------------------------------

Before calling this method, all transmission and receive activities have to be finished. This is responsibility of the programmer.

Note this method is NOT re-entrant. In other word, this member function can be called from both task and interrupt context.

Reimplemented from [murasaki::UartStrategy](#).

7.44.3.6 **murasaki::UartStatus** **murasaki::Uart::Transmit** ( *const uint8\_t \* data*, *unsigned int size*, *unsigned int timeout\_ms* ) [virtual]

## Parameters

<i>data</i>	Data buffer to be transmitted.
<i>size</i>	The count of the data ( byte ) to be transfered. Must be smaller than 65536
<i>timeout_ms</i>	Time out limit by milliseconds.

## Returns

True if all data transfered completely. False if time out happen.

Transmit given data buffer through an UART device.

The transmission mode is synchronous. That means, function returns when all data has been transmitted, except timeout. Passing [murasaki::kwmsIndefinitely](#) to the parameter `timeout_ms` orders not to return until complete transmission. Other value of `timeout_ms` parameter specifies the time out by millisecond. If time out happen, function returns false. If not happen, it returns true.

This function is exclusive. Internally the function is guarded by mutex. Then this function is thread safe. This function is forbidden to call from ISR.

Implements [murasaki::UartStrategy](#).

7.44.3.7 `bool murasaki::Uart::TransmitCompleteCallback ( void *const ptr ) [virtual]`

## Parameters

<i>ptr</i>	Pointer to UART_HandleTypeDef struct.
------------	---------------------------------------

## Returns

true: ptr matches with UART device and handle the call back. false : doesn't match.

A call back to notify the end of entire block transfer. This is considered as the end of DMA based transmission. The context have to be interrupt.

This member function checks whether the given ptr parameter matches its own device, and if matched, Release the waiting task and return true. If it doesn't match, just return false.

The return values are:

- [murasaki::kursOK](#) : Received complete.
- [murasaki::kursTimeOut](#) : Time out occur.
- [murasaki::kursFrame](#) : Receive error by wrong word size configuration.
- [murasaki::kursParity](#) : Parity error.
- [murasaki::kursNoise](#) : Error by noise.
- [murasaki::kursDMA](#) : This is fatal problem in HAL. Peripheral is re-initialized internally.
- other : This is fatal problem in HAL. Peripheral is re-initialized internally.

This method have to be called from [HAL\\_UART\\_TxCpltCallback\(\)](#). See STM32F7 HAL manual for detail



```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
    if (murasaki::platform.uart_console->TransmitCompleteCallback(huart))
        return;
}
```

Implements [murasaki::UartStrategy](#).

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

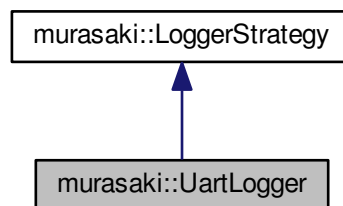
- [uart.hpp](#)
- [uart.cpp](#)

## 7.45 murasaki::UartLogger Class Reference

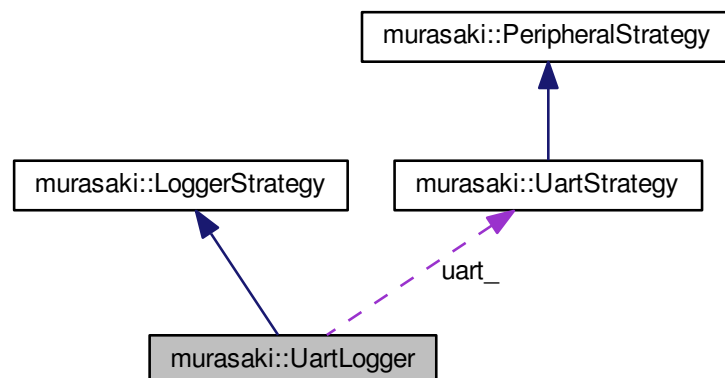
Logging through an UART port.

```
#include <uartlogger.hpp>
```

Inheritance diagram for murasaki::UartLogger:



Collaboration diagram for murasaki::UartLogger:



## Public Member Functions

- [UartLogger](#) ([UartStrategy](#) \*uart)  
*Constructor.*
- virtual void [putMessage](#) (char message[ ], unsigned int size)  
*Message output member function.*
- virtual char [getCharacter](#) ()  
*Character input member function.*
- virtual void [DoPostMortem](#) (void \*debugger\_fifo)  
*Start post mortem process.*

## Additional Inherited Members

### 7.45.1 Detailed Description

This is a standard logging class through the UART port. The instance of this class can be passed to the [murasaki::Debugger](#) constructor.

See [Application Specific Platform](#) as usage example.

### 7.45.2 Constructor & Destructor Documentation

#### 7.45.2.1 [murasaki::UartLogger::UartLogger](#) ( [UartStrategy](#) \* *uart* )

##### Parameters

<i>uart</i>	Pointer to the uart object.
-------------	-----------------------------

### 7.45.3 Member Function Documentation

#### 7.45.3.1 void [murasaki::UartLogger::DoPostMortem](#) ( void \* *debugger\_fifo* ) [virtual]

##### Parameters

<i>debugger_fifo</i>	Pointer to the <a href="#">DebuggerFifo</a> class object. The data inside this FIFO will be sent to UART This member function read the data in given FIFO, and then do the auto history.
----------------------	--

This function call the [DebuggerFifo::SetPostMortem\(\)](#) internally. Then, output the data inside FIFO through the given UART.

Once all the data is output, this function wait for a receive data. Once data received, this function rewind the FIFO and then, start to transmit the data again.

Reimplemented from [murasaki::LoggerStrategy](#).

#### 7.45.3.2 char [murasaki::UartLogger::getCharacter](#) ( ) [virtual]

**Returns**

A character from input is returned.

This function is considered as blocking and synchronous. That mean, the function will wait for any user input forever.

Implements [murasaki::LoggerStrategy](#).

**7.45.3.3** `void murasaki::UartLogger::putMessage ( char message[], unsigned int size )` `[virtual]`

**Parameters**

<i>message</i>	Non null terminated character array. This data is stored or output to the logger.
<i>size</i>	Size of the message[bytes]. Must be smaller than 65536

Implements [murasaki::LoggerStrategy](#).

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

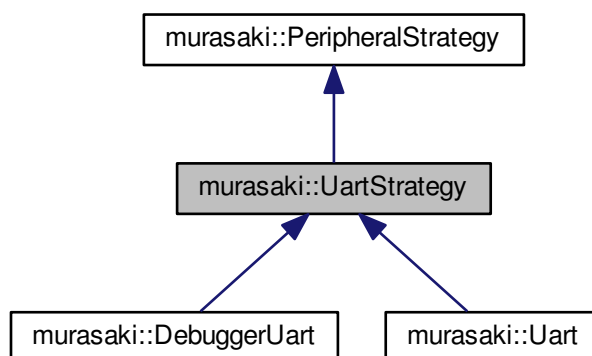
- [uartlogger.hpp](#)
- [uartlogger.cpp](#)

**7.46 murasaki::UartStrategy** Class Reference

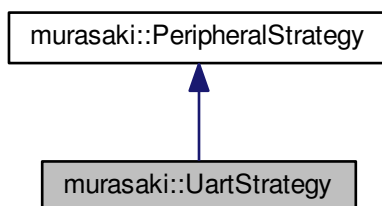
Definition of the root class of UART.

```
#include <uartstrategy.hpp>
```

Inheritance diagram for `murasaki::UartStrategy`:



Collaboration diagram for `murasaki::UartStrategy`:



### Public Member Functions

- virtual void `SetHardwareFlowControl` (`UartHardwareFlowControl` control)  
*Set the behavior of the hardware flow control.*
- virtual void `SetSpeed` (unsigned int speed)  
*the baud rate*
- virtual `murasaki::UartStatus Transmit` (const uint8\_t \*data, unsigned int size, unsigned int timeout\_ms=`murasaki::kwmsIndefinitely`)=0  
*buffer transmission over the UART. synchronous*
- virtual `murasaki::UartStatus Receive` (uint8\_t \*data, unsigned int size, unsigned int \*transferred\_count=nullptr, `UartTimeout` uart\_timeout=`murasaki::kutNoldleTimeout`, unsigned int timeout\_ms=`murasaki::kwmsIndefinitely`)=0  
*buffer receive over the UART. synchronous*
- virtual bool `TransmitCompleteCallback` (void \*ptr)=0  
*Call back to be called notify the transfer is complete.*
- virtual bool `ReceiveCompleteCallback` (void \*ptr)=0  
*Call back to be called for entire block transfer is complete.*
- virtual bool `HandleError` (void \*ptr)=0  
*Handling error report of device.*

### Additional Inherited Members

#### 7.46.1 Detailed Description

A prototype of the UART device. This abstract class shows the usage of the UART peripheral.

This prototype assumes the derived class will transmit / receive data in the task context on RTOS. And both methods should be synchronous. That means, until the transmit / receive terminates, both methods don't return.

Two callback methods are prepared to sync with the interrupt which tells the end of Transmit/Receive.

#### 7.46.2 Member Function Documentation

7.46.2.1 virtual bool `murasaki::UartStrategy::HandleError` ( void \* ptr ) [pure virtual]

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a device control
------------	---

## Returns

true if ptr matches with device and handle the error. false if ptr doesn't match A member function to detect error.

The error handling is depend on the implementation.

Implemented in [murasaki::Uart](#), and [murasaki::DebuggerUart](#).

**7.46.2.2** `virtual murasaki::UartStatus murasaki::UartStrategy::Receive ( uint8_t * data, unsigned int size, unsigned int * transfered_count = nullptr, UartTimeout uart_timeout = murasaki::kutNoldleTimeout, unsigned int timeout_ms = murasaki::kwmsIndefinitely ) [pure virtual]`

## Parameters

<i>data</i>	Pointer to the buffer to save the received data.
<i>size</i>	Number of the data to be received.
<i>transfered_count</i>	Number of bytes transfered. The nullptr means no need to return value.
<i>uart_timeout</i>	Specify <a href="#">murasaki::kutIdleTimeout</a> , if idle line timeout is needed.
<i>timeout_ms</i>	Time out by milli Second.

## Returns

Status of the IO processing

Implemented in [murasaki::Uart](#), and [murasaki::DebuggerUart](#).

**7.46.2.3** `virtual bool murasaki::UartStrategy::ReceiveCompleteCallback ( void * ptr ) [pure virtual]`

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a UART device control
------------	--

## Returns

true: ptr matches with UART device and handle the call back. false : doesn't match.

A call back to notify the end of entire block or byte transfer. The definition of calling timing is depend on the implementation. This is called from an DMA ISR.

Typically, an implementation may check whether the given ptr parameter matches its own device, and if matched, handle it and return true. If it doesn't match, just return false.

Implemented in [murasaki::Uart](#), and [murasaki::DebuggerUart](#).

**7.46.2.4** `virtual void murasaki::UartStrategy::SetHardwareFlowControl ( UartHardwareFlowControl control ) [inline], [virtual]`

## Parameters

<i>control</i>	The control mode.
----------------	-------------------

Reimplemented in [murasaki::Uart](#), and [murasaki::DebuggerUart](#).

7.46.2.5 `virtual void murasaki::UartStrategy::SetSpeed ( unsigned int speed ) [inline],[virtual]`

## Parameters

<i>speed</i>	BAUD rate ( 110, 300, ... 9600,... )
--------------	--------------------------------------

Reimplemented in [murasaki::Uart](#), and [murasaki::DebuggerUart](#).

7.46.2.6 `virtual murasaki::UartStatus murasaki::UartStrategy::Transmit ( const uint8_t * data, unsigned int size, unsigned int timeout_ms = murasaki::kwmsIndefinitely ) [pure virtual]`

## Parameters

<i>data</i>	Pointer to the buffer to be sent.
<i>size</i>	Number of the data to be sent.
<i>timeout_ms</i>	Time out by mili Second.

## Returns

Status of the IO processing

Implemented in [murasaki::Uart](#), and [murasaki::DebuggerUart](#).

7.46.2.7 `virtual bool murasaki::UartStrategy::TransmitCompleteCallback ( void * ptr ) [pure virtual]`

## Parameters

<i>ptr</i>	Pointer for generic use. Usually, points a struct of a UART device control
------------	--

## Returns

true: ptr matches with UART device and handle the call back. false : doesn't match.

A call back to notify the end of entire block or byte transfer. The definition of calling timing is depend on the implementation. This is called from an DMA ISR.

Typically, an implementation may check whether the given ptr parameter matches its own device, and if matched, handle it and return true. If it doesn't match, just return false.

Implemented in [murasaki::Uart](#), and [murasaki::DebuggerUart](#).

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

- [uartstrategy.hpp](#)

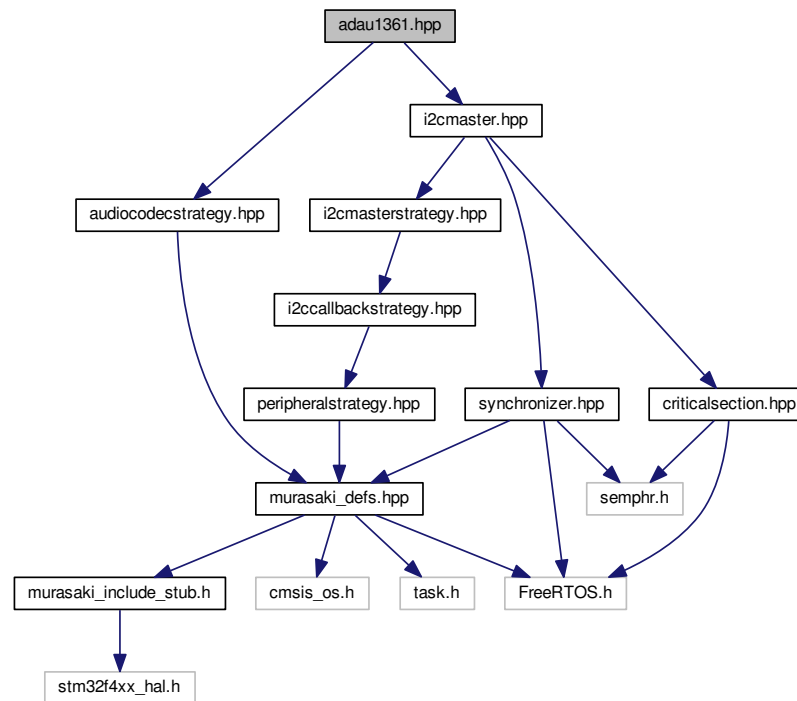
## 8 File Documentation

### 8.1 adau1361.hpp File Reference

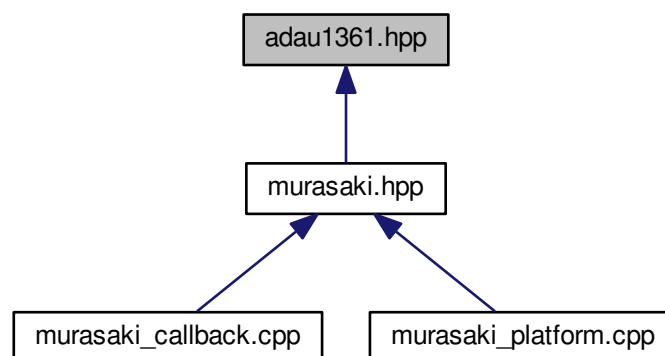
```
#include <audiocodecstrategy.hpp>
```

```
#include "i2cmaster.hpp"
```

Include dependency graph for adau1361.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::Adau1361](#)  
*Audio Codec LSI class.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

## 8.1.1 Detailed Description

## Date

2018/05/11

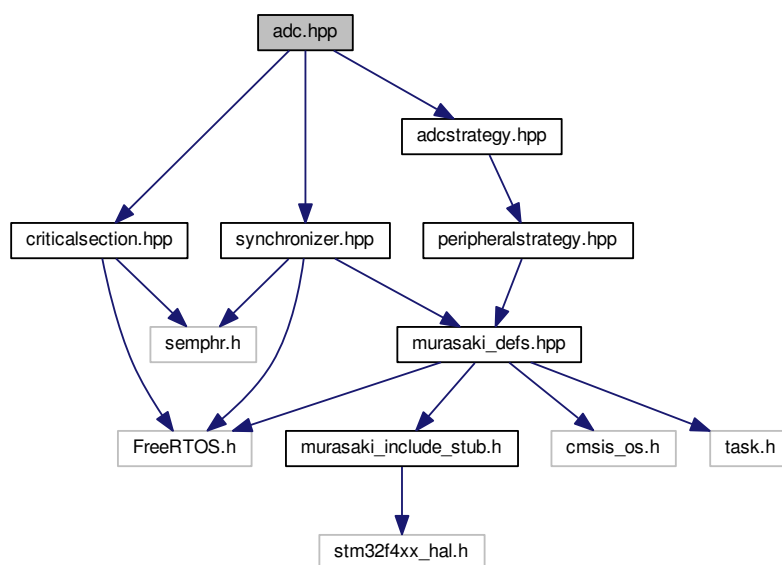
## Author

: Seiichi "Suikan" Horie

8.2 `adc.hpp` File Reference

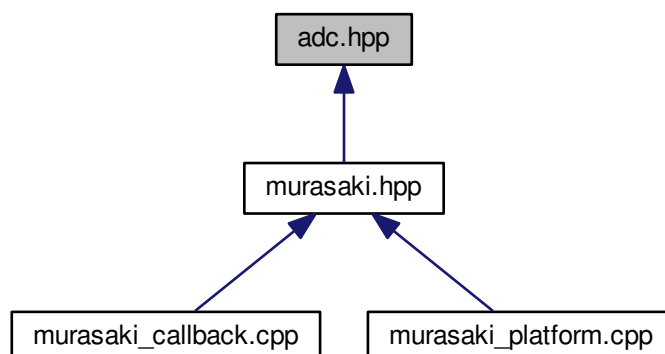
AD Coverter class.

```
#include "synchronizer.hpp"
#include "criticalsection.hpp"
#include "adcstrategy.hpp"
Include dependency graph for adc.hpp:
```





This graph shows which files directly or indirectly include this file:



### Classes

- class `murasaki::Adc`  
*STM32 dedicated ADC class.*

### Namespaces

- `murasaki`  
*Personal [Platform](#) parts collection.*

#### 8.2.1 Detailed Description

#### Date

Feb 22, 2020

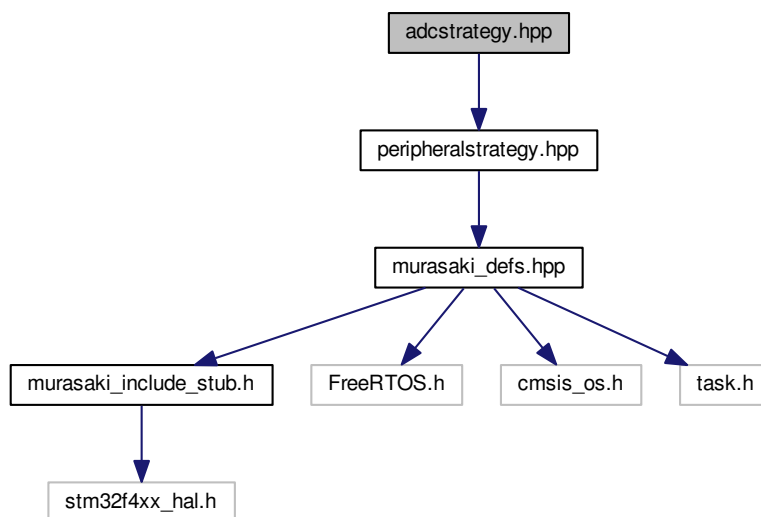
#### Author

Seiichi Horie

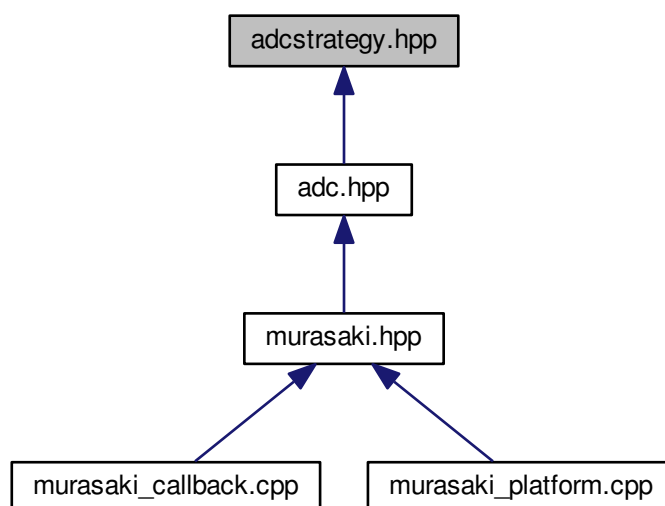
### 8.3 adcstrategy.hpp File Reference

```
#include <peripheralstrategy.hpp>
```

Include dependency graph for adcstrategy.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::AdcStrategy](#)

*Synchronized, blocking ADC converter Strategy.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.3.1 Detailed Description

#### Date

Feb 22, 2020

#### Author

Seiichi Horie

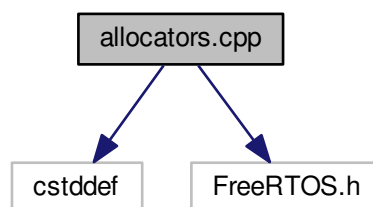
## 8.4 allocators.cpp File Reference

Alternative memory allocators.

```
#include <cstddef>
```

```
#include <FreeRTOS.h>
```

Include dependency graph for allocators.cpp:



## Functions

- void \* [operator new](#) (std::size\_t size)  
*Allocate a memory piece with given size.*
- void \* [operator new\[\]](#) (std::size\_t size)  
*Allocate a memory piece with given size.*
- void [operator delete](#) (void \*ptr)  
*Deallocate the given memory.*
- void [operator delete\[\]](#) (void \*ptr)  
*Deallocate the given memory.*

### 8.4.1 Detailed Description

#### Date

2018/05/02

#### Author

Seiichi "Suikan" Horie

These definitions allows to used the FreeRTOS's heap instead of the system heap.

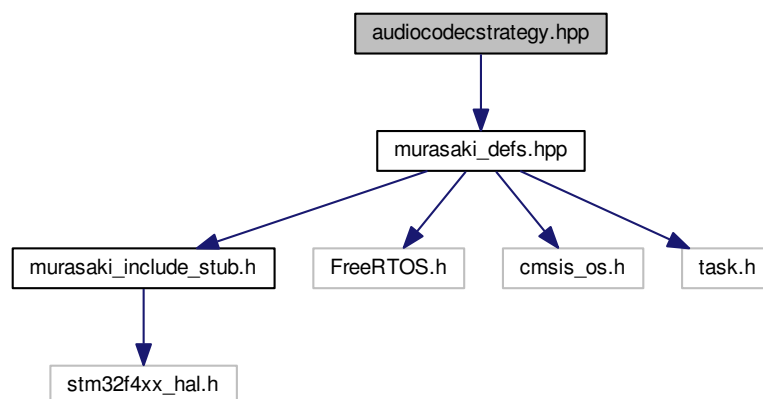
The system heap by the standard library doesn't check the limit of the heap cerefly. As a result, it is not clear how to detect the over committing memory.

FreeRTOS hepa is considered safer than system heap. Then, the new and the delete operators are overloaded to use the pvPortMalloc().

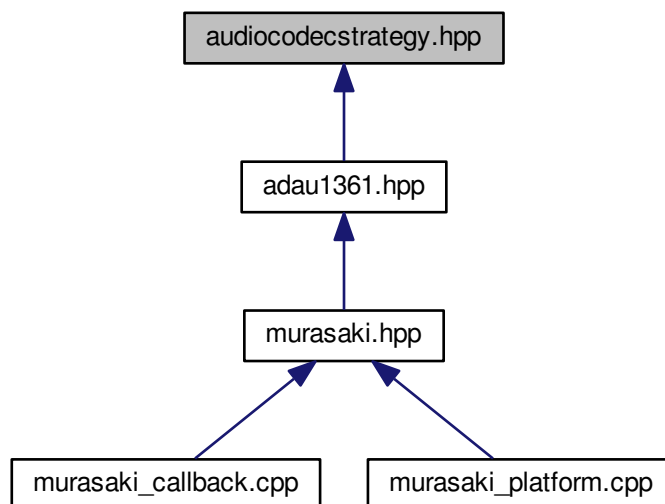
## 8.5 audiocodecstrategy.hpp File Reference

```
#include <murasaki_defs.hpp>
```

Include dependency graph for audiocodecstrategy.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::AudioCodecStrategy](#)  
*abstract audio CODEC controller.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.5.1 Detailed Description

##### Date

2018/05/11

##### Author

: Seiichi "Suikan" Horie

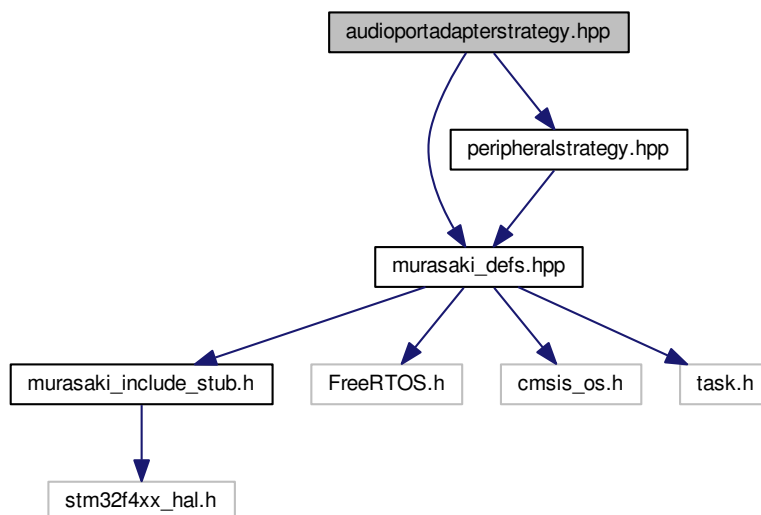
## 8.6 audioportadapterstrategy.hpp File Reference

Strategy of the Audio device adaptor.

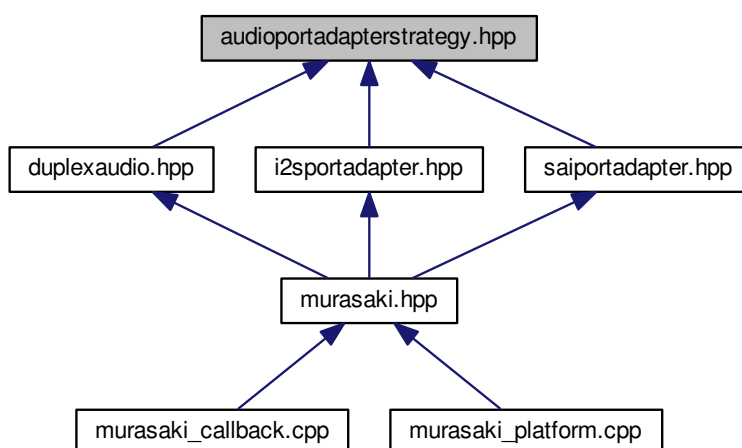
```
#include "murasaki_defs.hpp"
```

```
#include "peripheralstrategy.hpp"
```

Include dependency graph for audioportadapterstrategy.hpp:



This graph shows which files directly or indirectly include this file:



### Classes

- class `murasaki::AudioPortAdapterStrategy`  
Strategy of the audio device adaptor..

## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

## 8.6.1 Detailed Description

## Date

2019/07/28

## Author

Seiichi "Suikan" Horie

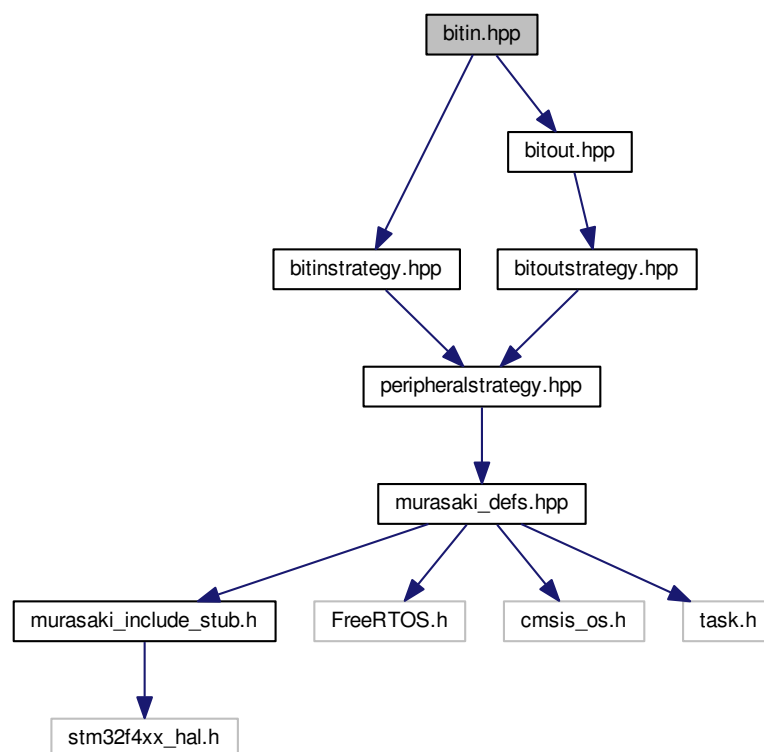
## 8.7 bitin.hpp File Reference

GPIO bit in class.

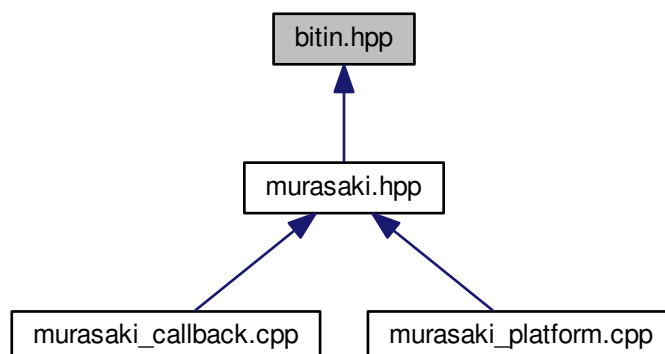
```
#include <bitinstrategy.hpp>
```

```
#include "bitout.hpp"
```

Include dependency graph for bitin.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::BitIn](#)  
*General purpose bit input.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.7.1 Detailed Description

##### Date

2018/05/07

##### Author

Seiichi "Suikan" Horie

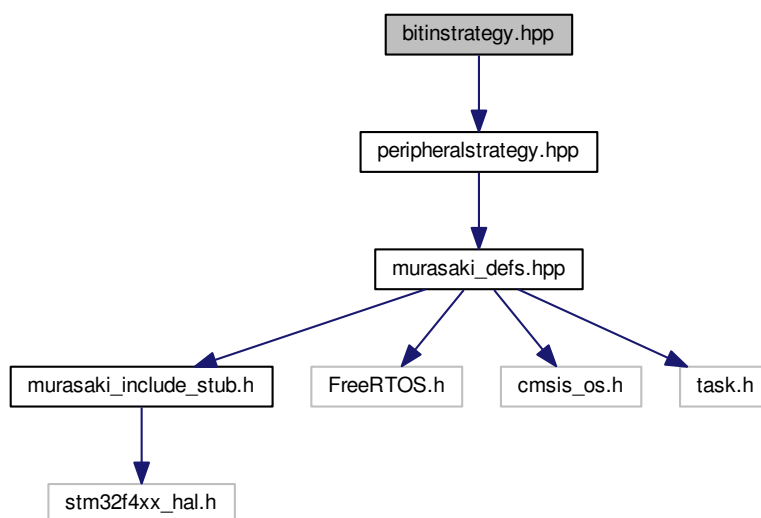
#### 8.8 bitinstrategy.hpp File Reference

Abstract class of the GPIO bit in.

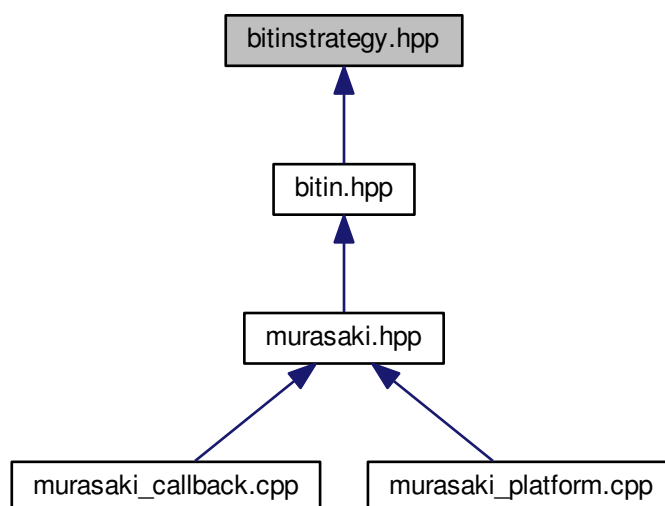


```
#include <peripheralstrategy.hpp>
```

Include dependency graph for bitinstrategy.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::BitInStrategy](#)  
*Definition of the root class of bit input.*

## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

### 8.8.1 Detailed Description

#### Date

2018/05/07

#### Author

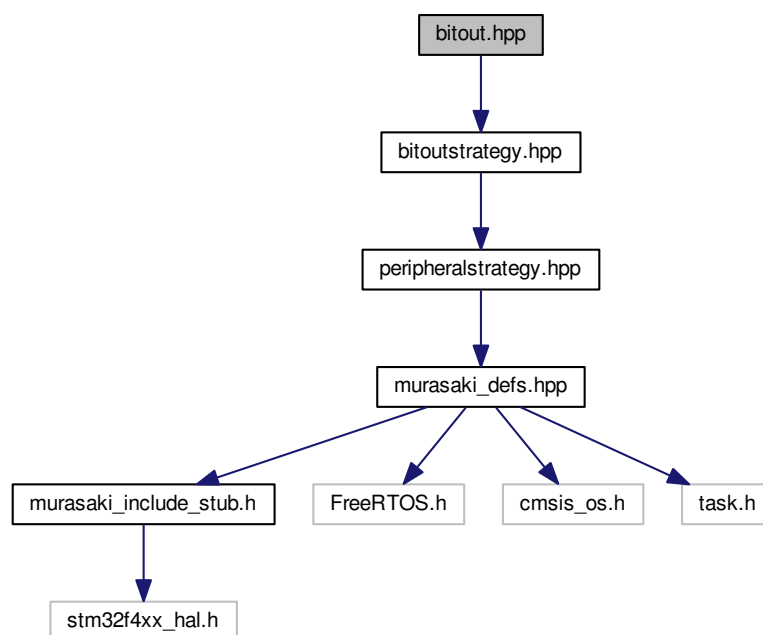
Seiichi "Suikan" Horie

## 8.9 bitout.hpp File Reference

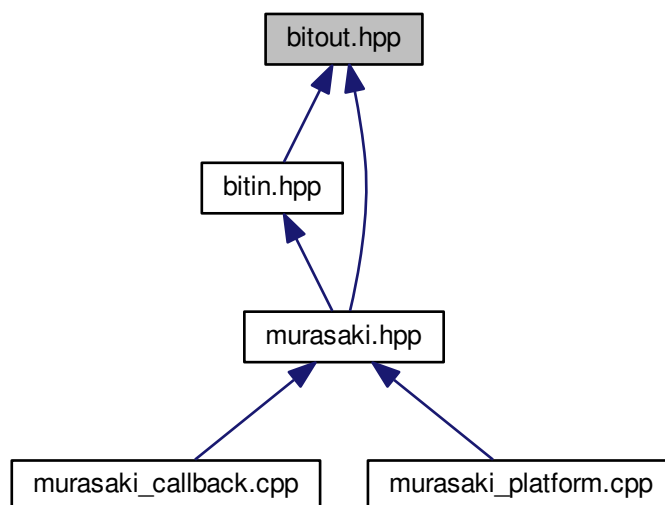
GPIO bit out class.

```
#include <bitoutstrategy.hpp>
```

Include dependency graph for bitout.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- struct [murasaki::GPIO\\_type](#)  
*A structure to en-group the GPIO port and GPIO pin.*
- class [murasaki::BitOut](#)  
*General purpose bit output.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.9.1 Detailed Description

##### Date

2018/05/07

##### Author

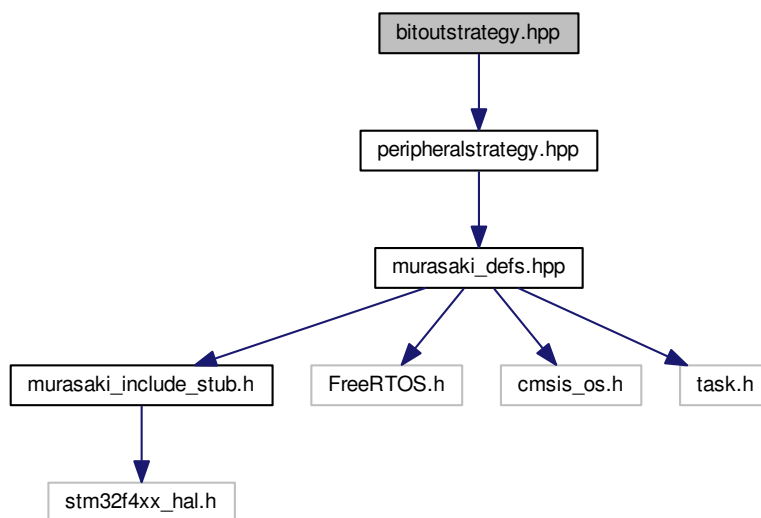
Seiichi "Suikan" Horie

## 8.10 bitoutstrategy.hpp File Reference

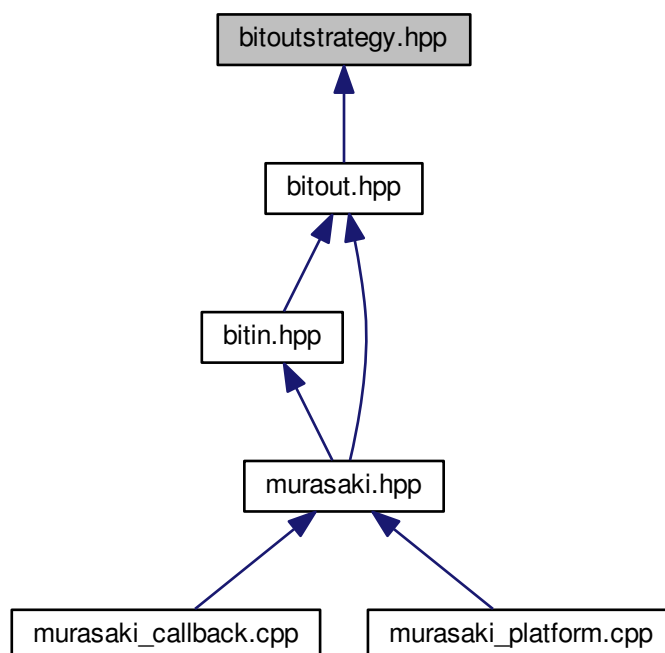
Abstract class of GPIO bit out.

```
#include <peripheralstrategy.hpp>
```

Include dependency graph for bitoutstrategy.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::BitOutStrategy](#)  
*Definition of the root class of bit output.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.10.1 Detailed Description

##### Date

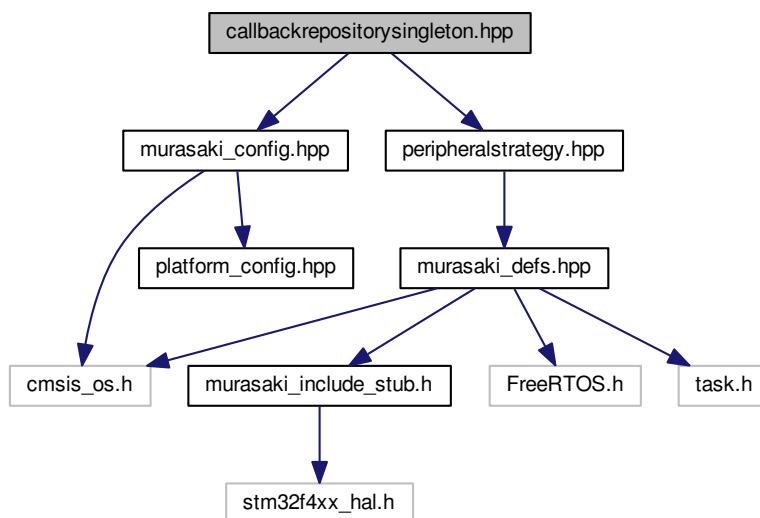
2018/05/07

##### Author

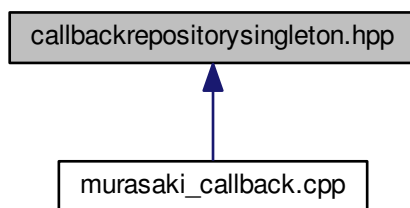
Seiichi "Suikan" Horie

## 8.11 callbackrepositorysingleton.hpp File Reference

```
#include "murasaki_config.hpp"
#include "peripheralstrategy.hpp"
Include dependency graph for callbackrepositorysingleton.hpp:
```



This graph shows which files directly or indirectly include this file:



### Classes

- class [murasaki::CallbackRepositorySingleton](#)  
*Repository of the interrupt handling objects.*

### Namespaces

- [murasaki](#)  
*Personal Platform parts collection.*

### 8.11.1 Detailed Description

Date

2020/04/11

## Author

takemasa

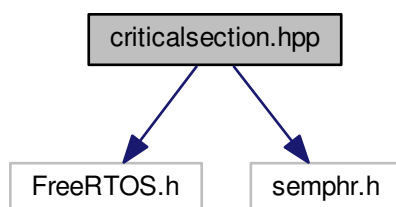
## 8.12 criticalsection.hpp File Reference

Class to protect a certain section from the interference.

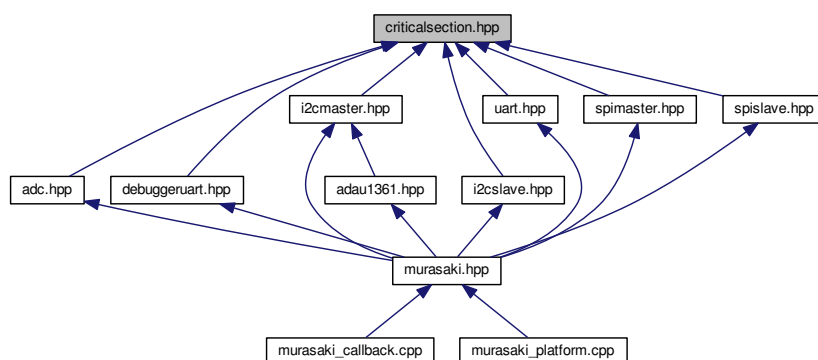
```
#include <FreeRTOS.h>
```

```
#include <semphr.h>
```

```
# Include dependency graph for criticalsection.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class `murasaki::CriticalSection`  
*A critical section for task context.*

## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

### 8.12.1 Detailed Description

#### Date

2018/01/27

#### Author

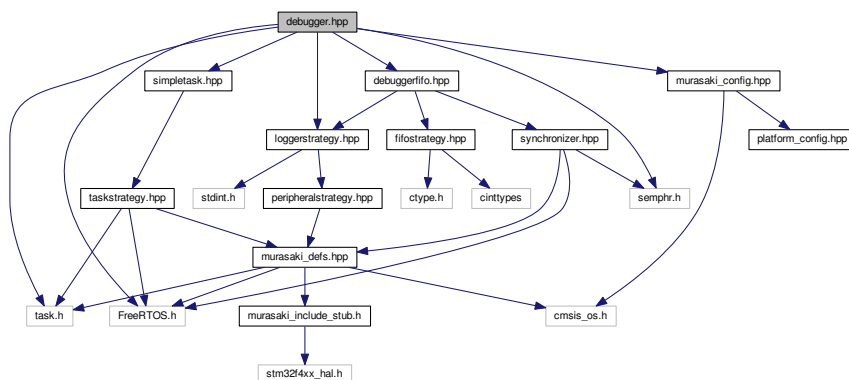
Seiichi "Suikan" Horie

## 8.13 debugger.hpp File Reference

Debug print class. For both ISR and task.

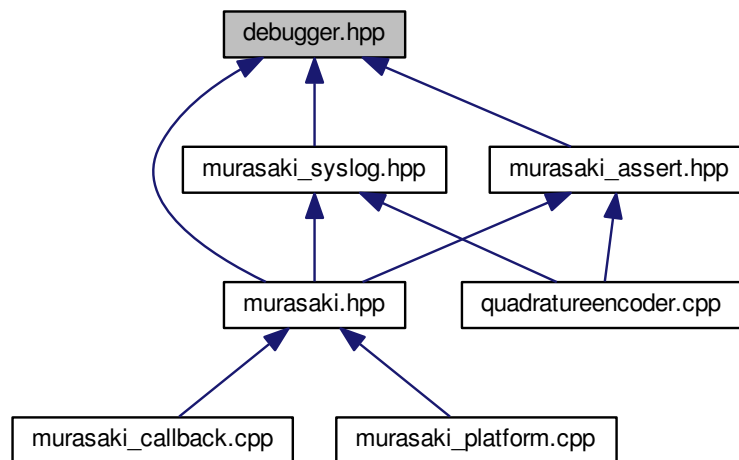
```
#include <FreeRTOS.h>
#include <loggerstrategy.hpp>
#include <task.h>
#include <semphr.h>
#include "murasaki_config.hpp"
#include "debuggerfifo.hpp"
#include "simpletask.hpp"
```

Include dependency graph for debugger.hpp:





This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::Debugger](#)  
*Debug class. Provides printf() style output for both task and ISR context.*

#### Namespaces

- [murasaki](#)  
*Personal Platform parts collection.*

#### Variables

- Debugger \* [murasaki::debugger](#)  
*A global variable to provide the debugging function.*

#### 8.13.1 Detailed Description

##### Date

2018/01/03

##### Author

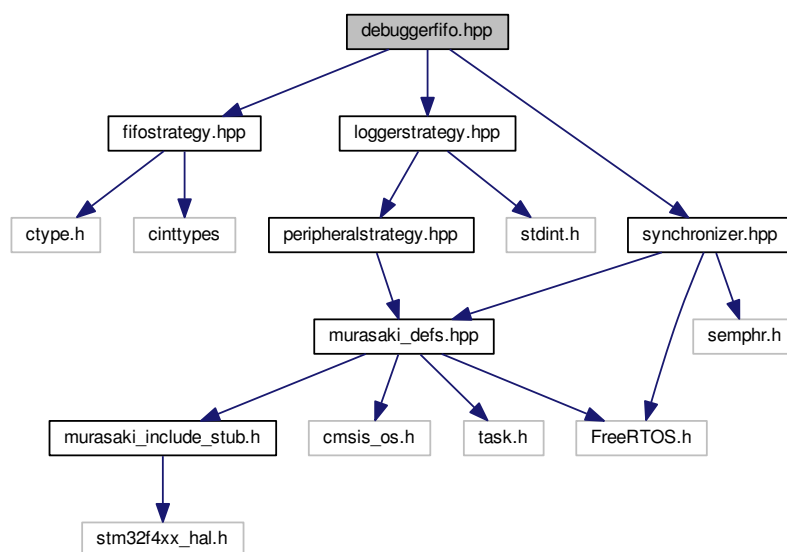
Seiichi "Suikan" Horie

This class serves printf function for both task context and ISR context.

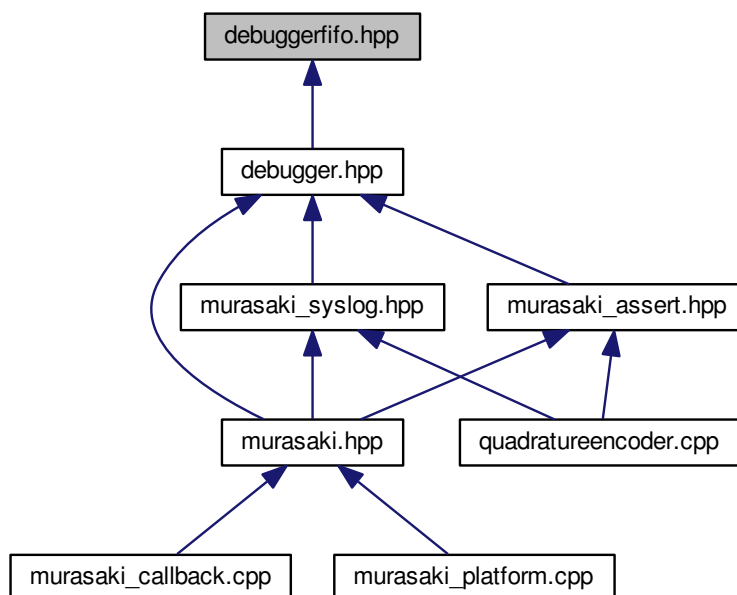
## 8.14 debuggerfifo.hpp File Reference

Dedicated FIFO to logging the debug message.

```
#include <fifostrategy.hpp>
#include <loggerstrategy.hpp>
#include "synchronizer.hpp"
Include dependency graph for debuggerfifo.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::DebuggerFifo](#)  
*FIFO with thread safe.*
- struct [murasaki::LoggingHelpers](#)  
*A stracture to engroup the logging tools.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.14.1 Detailed Description

#### Date

2018/03/01

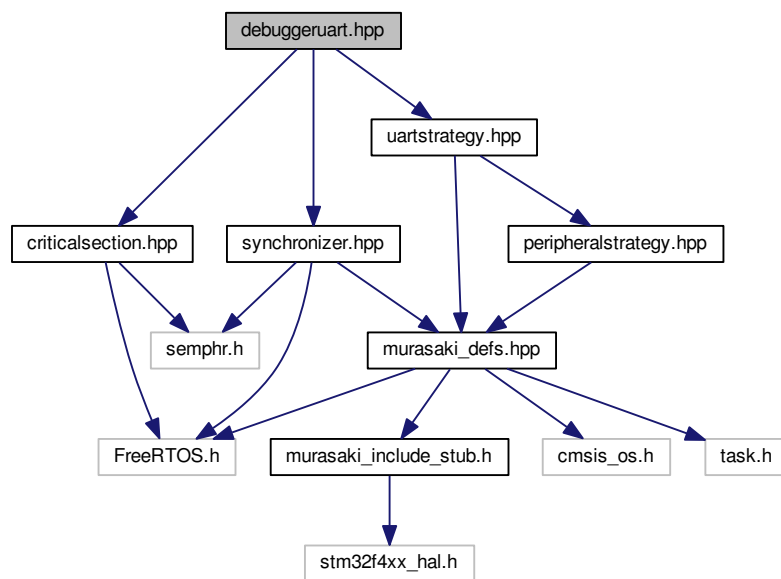
#### Author

Seiichi "Suikan" Horie

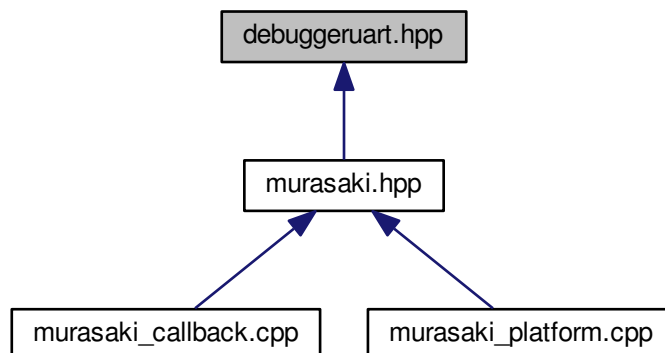
## 8.15 debuggeruart.hpp File Reference

UART. Thread safe and synchronous IO.

```
#include <synchronizer.hpp>
#include <uartstrategy.hpp>
#include "criticalsection.hpp"
Include dependency graph for debuggeruart.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::DebuggerUart](#)  
*Logging dedicated UART class.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

## 8.15.1 Detailed Description

## Date

2018/09/23

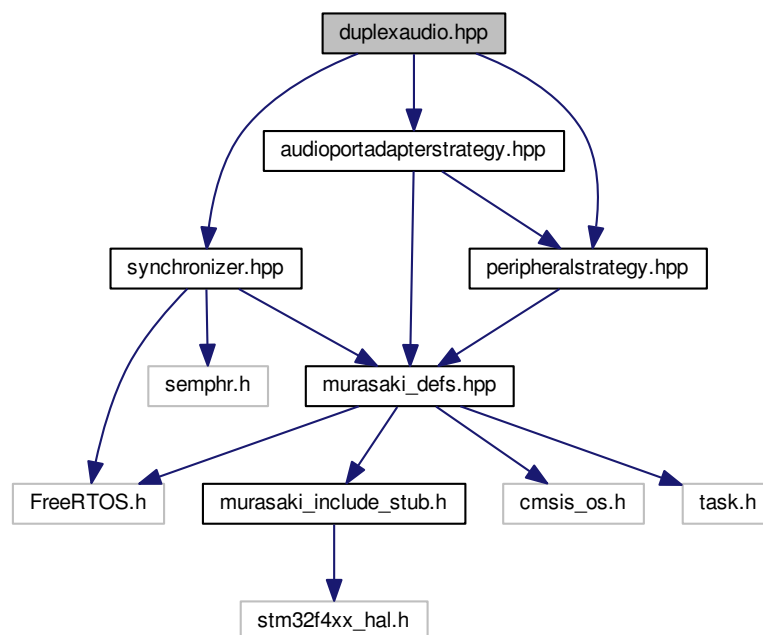
## Author

Seiichi "Suikan" Horie

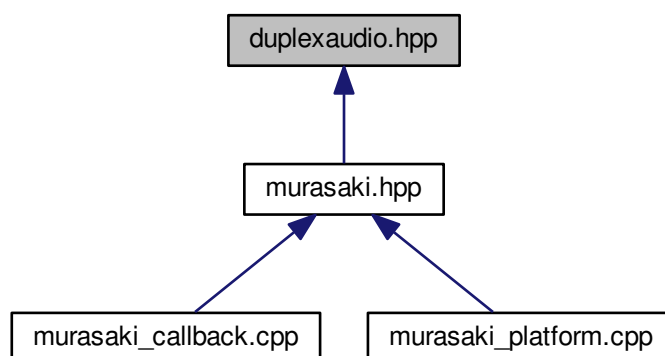
## 8.16 duplexaudio.hpp File Reference

root class of the stereo audio

```
#include "synchronizer.hpp"
#include "audioportadapterstrategy.hpp"
#include "peripheralstrategy.hpp"
Include dependency graph for duplexaudio.hpp:
```



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::DuplexAudio](#)  
*Stereo Audio is served by this class.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.16.1 Detailed Description

##### Date

2019/03/02

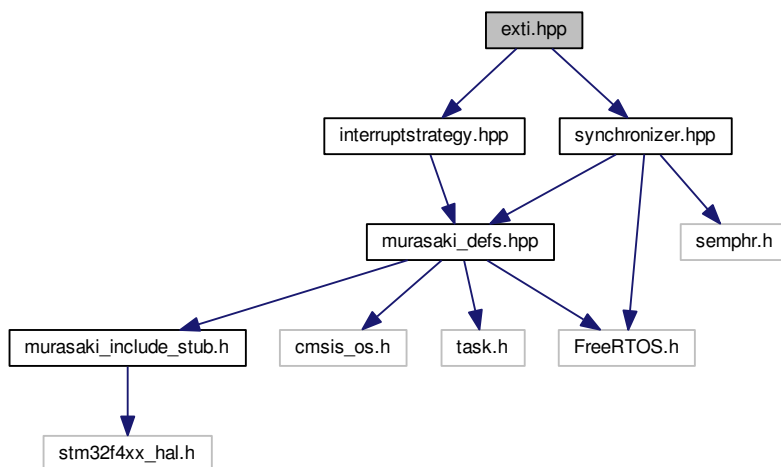
##### Author

Seiichi "Suikan" Horie

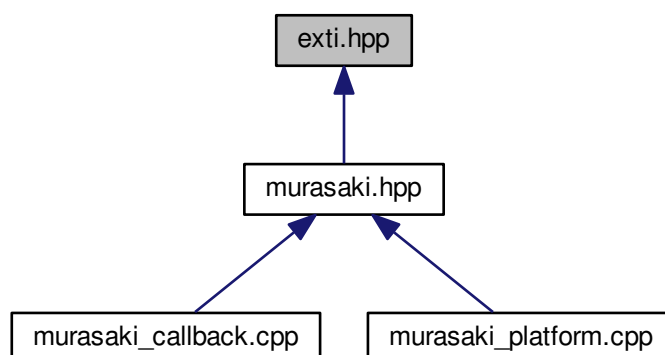
#### 8.17 exti.hpp File Reference

```
#include "interruptstrategy.hpp"  
#include "synchronizer.hpp"
```

Include dependency graph for exti.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::Exti](#)  
*EXTI wrapper class.*

## Namespaces

- [murasaki](#)  
*Personal Platform parts collection.*

### 8.17.1 Detailed Description

#### Date

Feb 29, 2020

#### Author

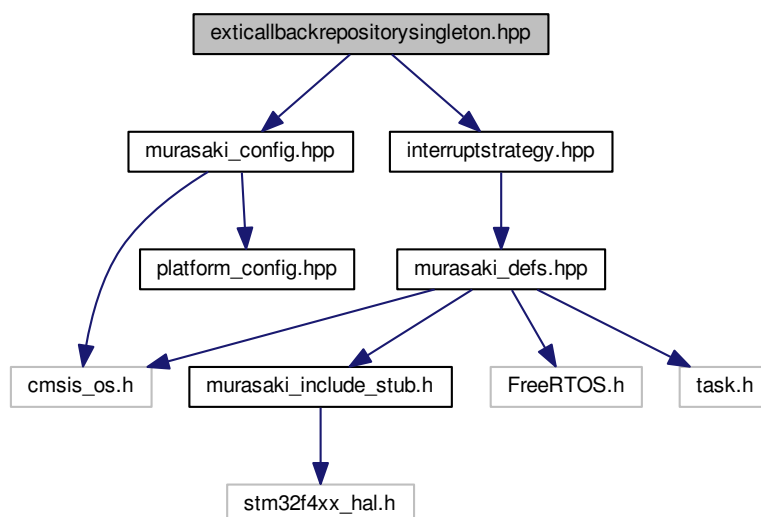
takemasa

## 8.18 extcallbackrepositorysingleton.hpp File Reference

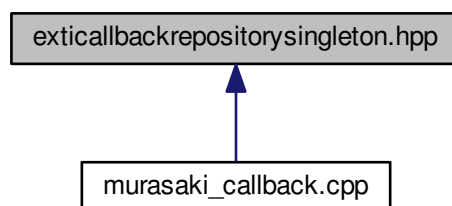
```
#include "murasaki_config.hpp"
```

```
#include "interruptstrategy.hpp"
```

Include dependency graph for extcallbackrepositorysingleton.hpp:



This graph shows which files directly or indirectly include this file:





## Classes

- class [murasaki::ExtiCallbackRepositorySingleton](#)  
*Repository of the EXTI handling objects.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.18.1 Detailed Description

#### Date

2020/04/18

#### Author

takemasa

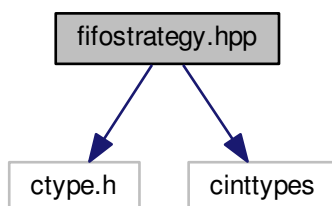
## 8.19 fifostrategy.hpp File Reference

Abstract class of FIFO.

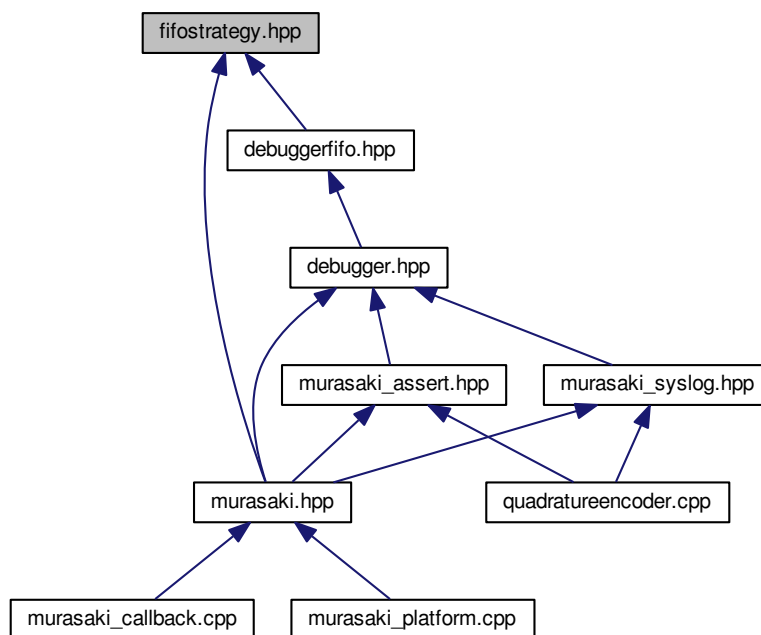
```
#include <ctype.h>
```

```
#include <cinttypes>
```

Include dependency graph for fifostrategy.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::FifoStrategy](#)  
*Basic FIFO without thread safe.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.19.1 Detailed Description

#### Date

2018/02/26

#### Author

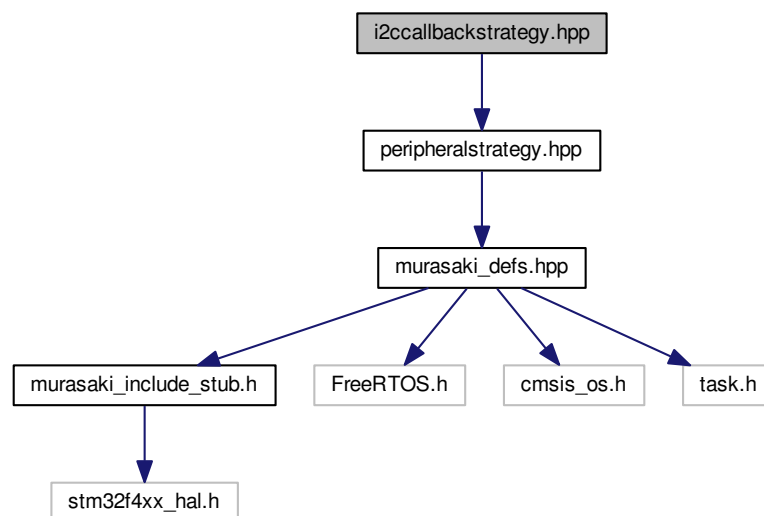
Seiichi "Suikan" Horie

## 8.20 i2ccallbackstrategy.hpp File Reference

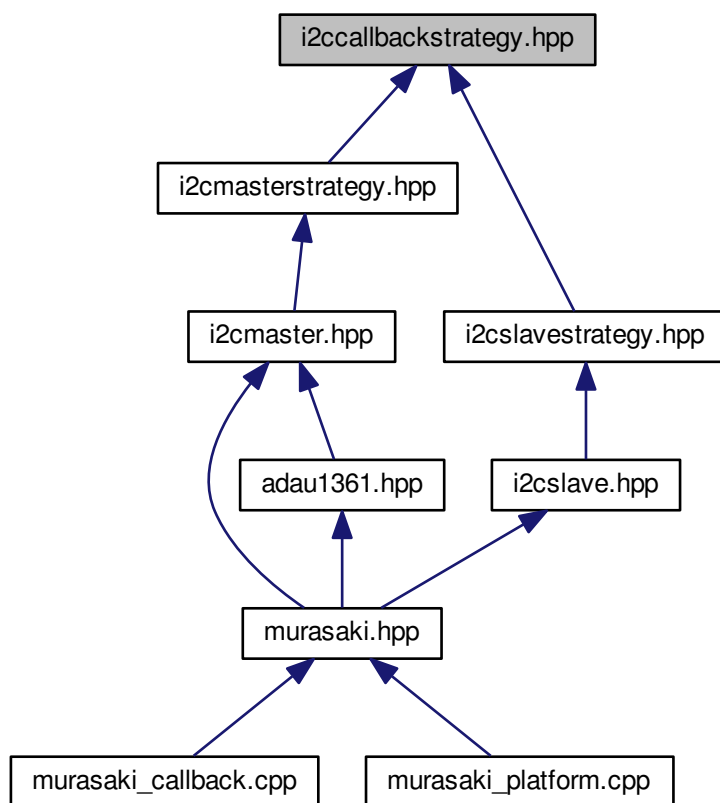
Root class definition of the I2C Master.

```
#include <peripheralstrategy.hpp>
```

Include dependency graph for i2ccallbackstrategy.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::I2cCallbackStrategy](#)  
*Definition of the root class of I2C .*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

## 8.20.1 Detailed Description

### Date

2018/02/11

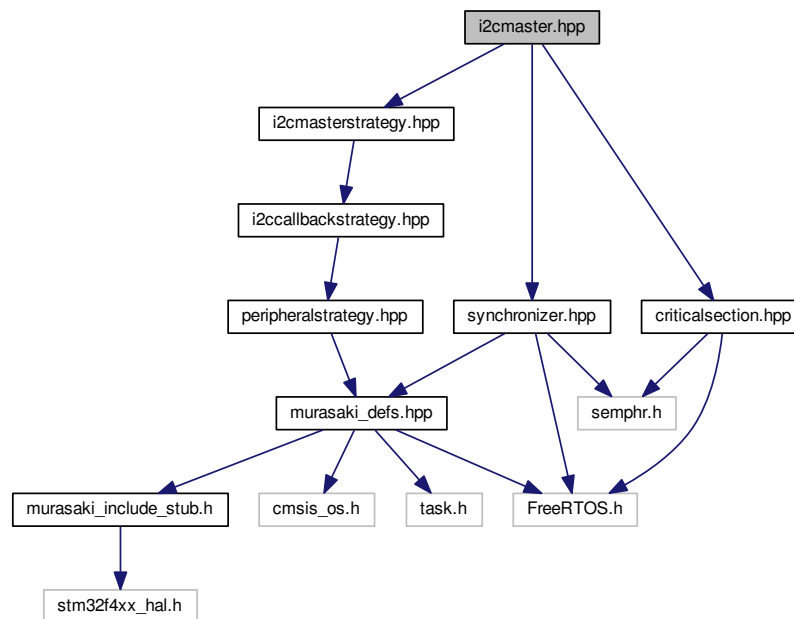
### Author

: Seiichi "Suikan" Horie

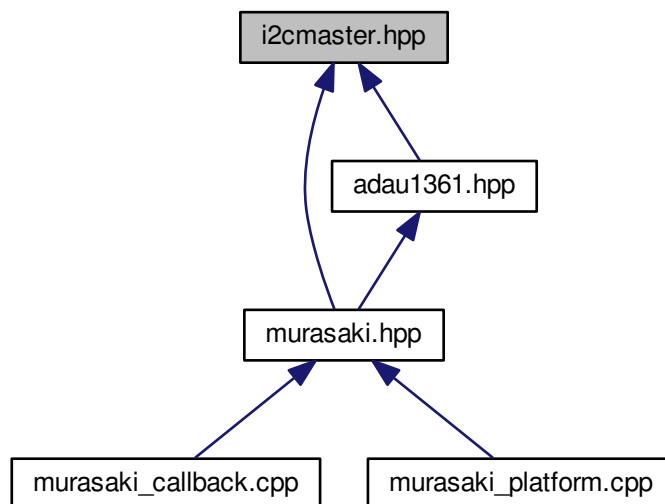
## 8.21 i2cmaster.hpp File Reference

I2C master. Thread safe, synchronous IO.

```
#include <i2cmasterstrategy.hpp>
#include <synchronizer.hpp>
#include "criticalsection.hpp"
Include dependency graph for i2cmaster.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::I2cMaster](#)

*Thread safe, synchronous, and blocking IO. Encapsulating I2C master. Based on STM32Cube HAL driver and FreeRTOS.*

## Namespaces

- [murasaki](#)

*Personal Platform parts collection.*

### 8.21.1 Detailed Description

#### Date

2018/02/12

#### Author

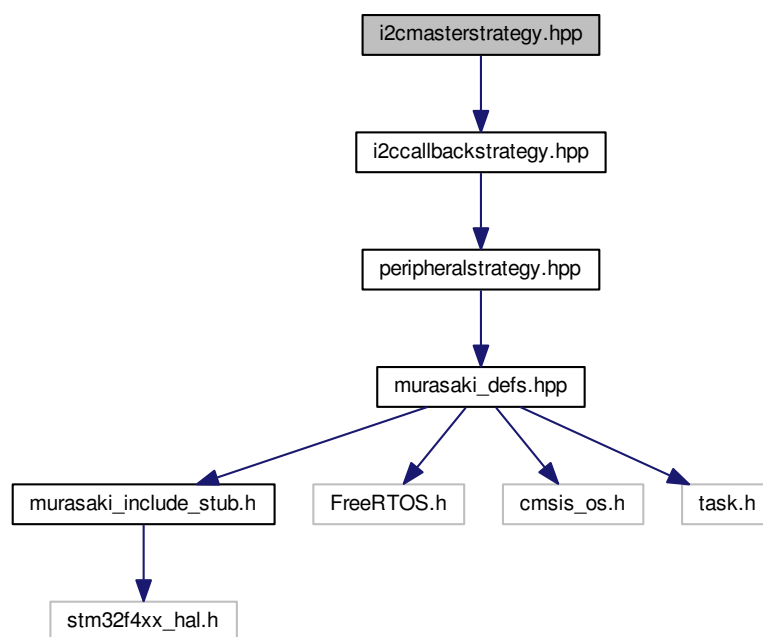
: Seiichi "Suikan" Horie

## 8.22 i2cmasterstrategy.hpp File Reference

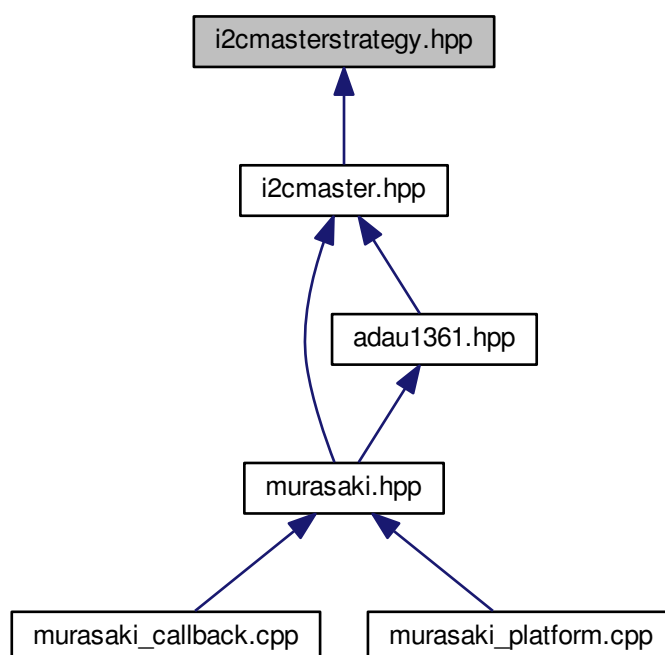
Root class definition of the I2C Master.

```
#include "i2ccallbackstrategy.hpp"
```

Include dependency graph for i2cmasterstrategy.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::I2cMasterStrategy](#)  
*Definition of the root class of I2C master.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.22.1 Detailed Description

#### Date

2018/02/11

#### Author

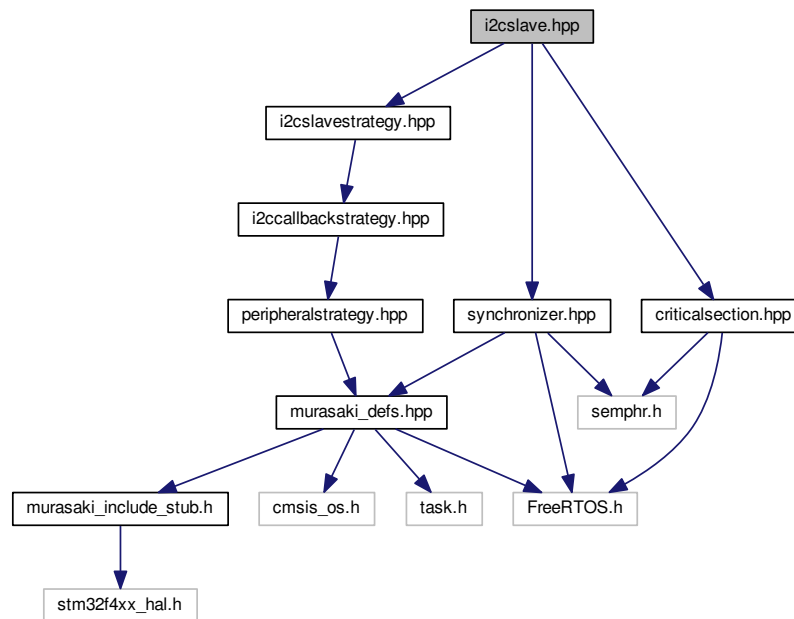
: Seiichi "Suikan" Horie



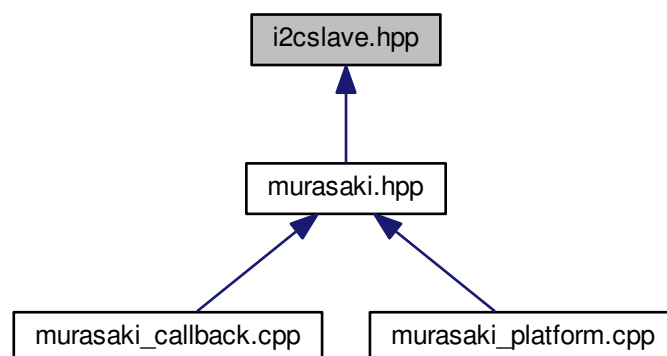
## 8.23 i2cslave.hpp File Reference

I2C slave. Thread safe, synchronous IO.

```
#include <i2cslavestrategy.hpp>
#include <synchronizer.hpp>
#include "criticalsection.hpp"
Include dependency graph for i2cslave.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::I2cSlave](#)

*Thread safe, synchronous and blocking IO. Encapsulating I2C slave. Based on STM32Cube HAL driver and FreeRTOS.*

## Namespaces

- [murasaki](#)

*Personal [Platform](#) parts collection.*

## 8.23.1 Detailed Description

## Date

2018/10/07

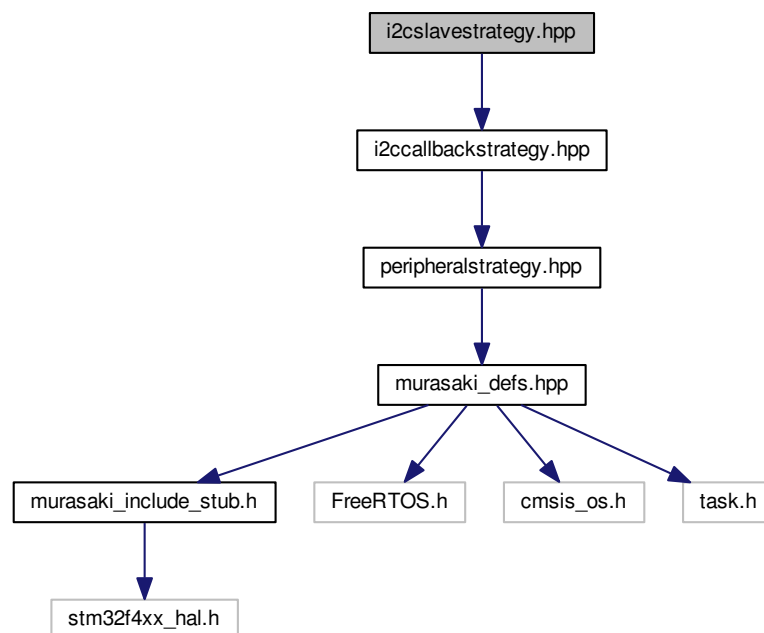
## Author

Seiichi "Suikan" Horie

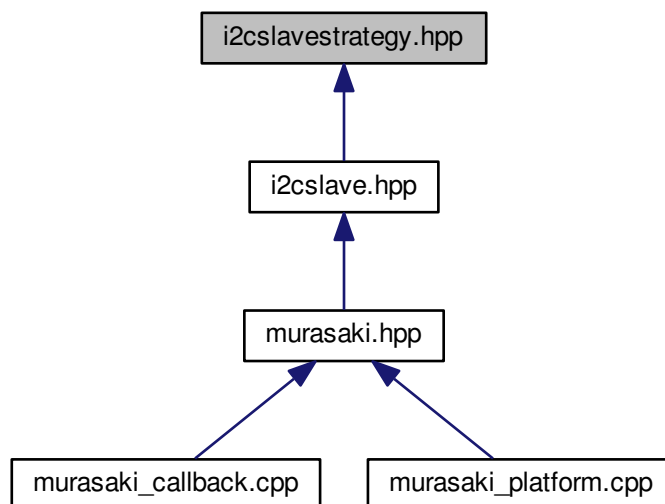
## 8.24 i2cslavestrategy.hpp File Reference

Root class definition of the I2C Slave.

```
#include "i2ccallbackstrategy.hpp"
Include dependency graph for i2cslavestrategy.hpp:
```



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::I2cSlaveStrategy](#)  
*Definition of the root class of I2C Slave.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.24.1 Detailed Description

#### Date

2018/10/07

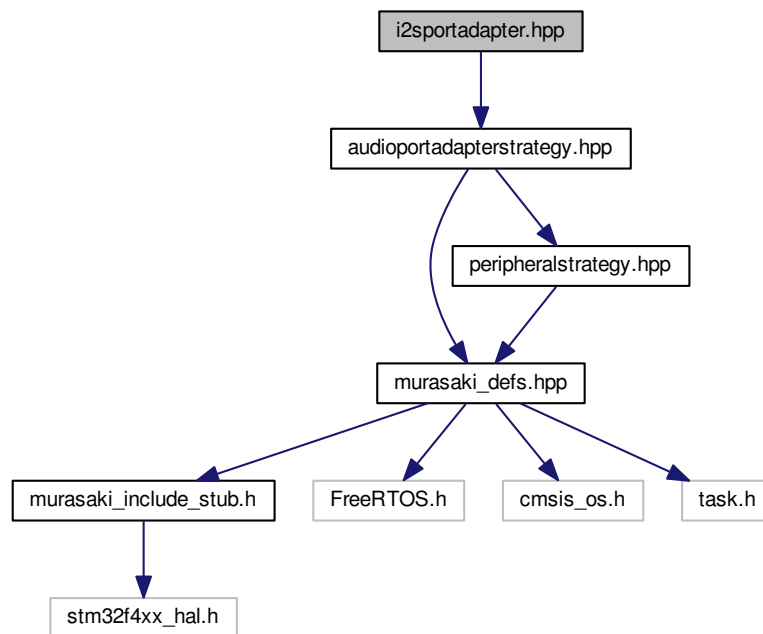
#### Author

Seiichi "Suikan" Horie

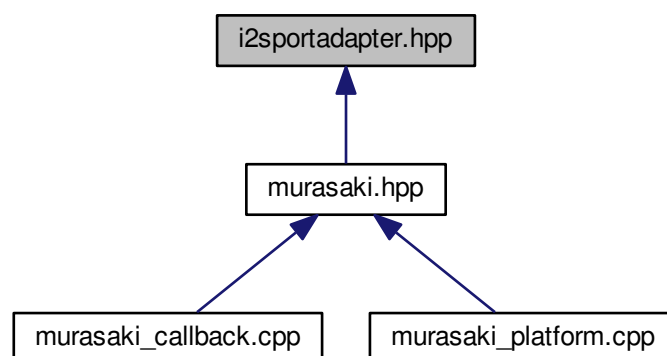
## 8.25 i2sportadapter.hpp File Reference

```
#include <audioportadapterstrategy.hpp>
```

Include dependency graph for i2sportadapter.hpp:



This graph shows which files directly or indirectly include this file:



### Classes

- class `murasaki::I2sPortAdapter`  
*Adapter as I2S audio port.*

## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

## 8.25.1 Detailed Description

## Date

2020/01/15

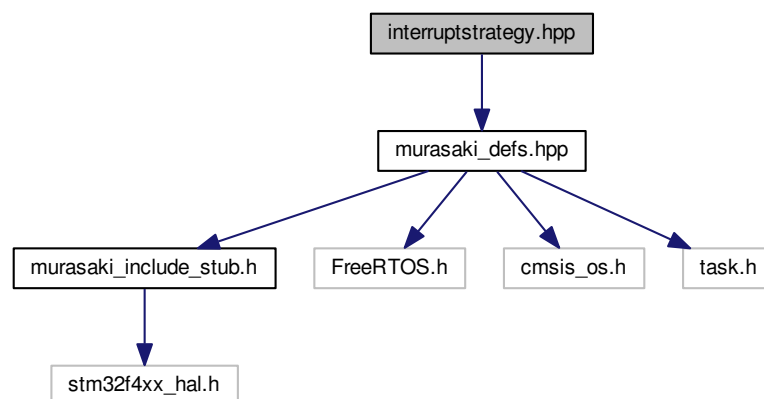
## Author

takemasa

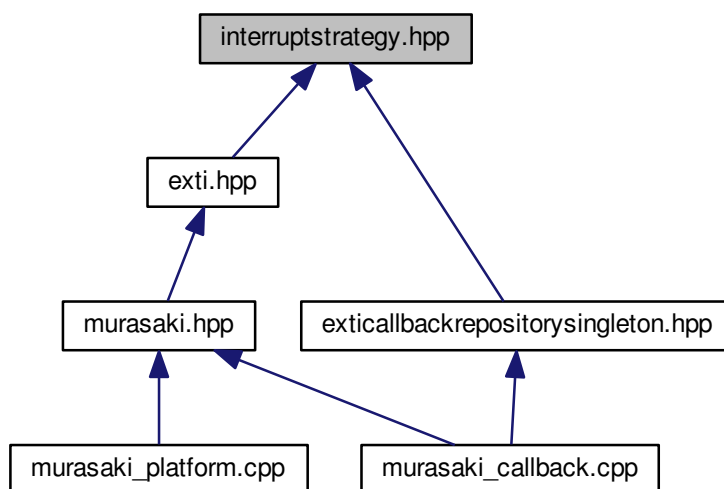
## 8.26 interruptstrategy.hpp File Reference

```
#include "murasaki_defs.hpp"
```

Include dependency graph for interruptstrategy.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::InterruptStrategy](#)  
*Abstract interrupt class.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.26.1 Detailed Description

#### Date

Feb 29, 2020

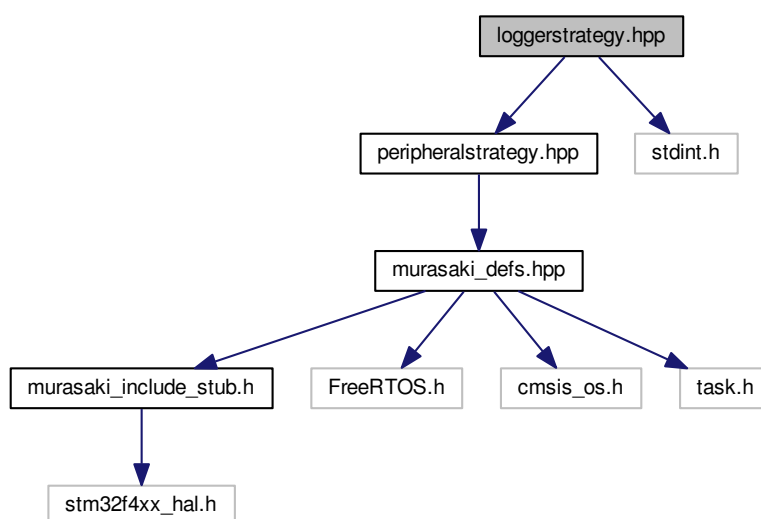
#### Author

Seiichi "Suikan" Horie

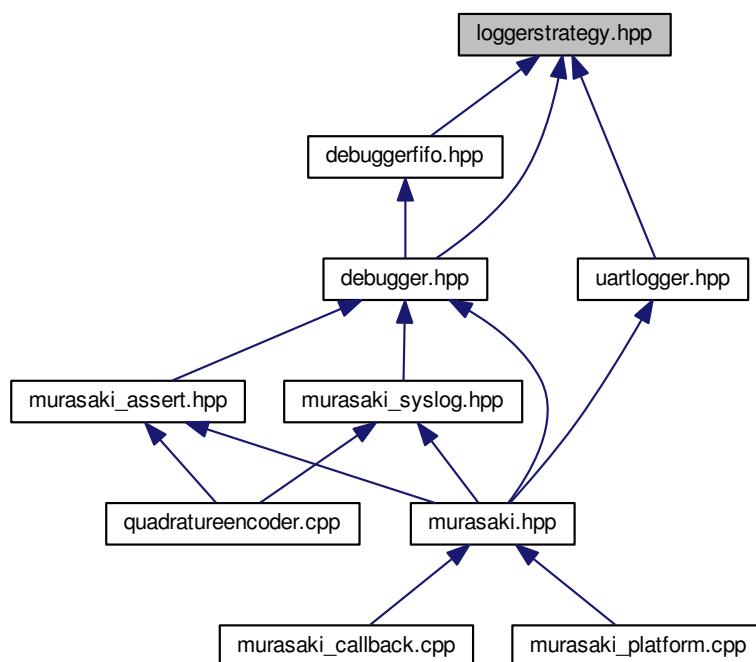
## 8.27 loggerstrategy.hpp File Reference

Simplified logging function.

```
#include <peripheralstrategy.hpp>
#include <stdint.h>
Include dependency graph for loggerstrategy.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::LoggerStrategy](#)  
*Abstract class for logging.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.27.1 Detailed Description

#### Date

2018/01/20

#### Author

: Seiichi "Suikan" Horie

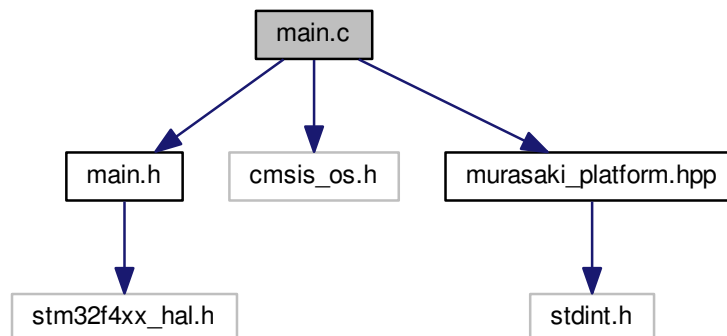


## 8.28 main.c File Reference

: Main program body

```
#include "main.h"
#include "cmsis_os.h"
#include "murasaki_platform.hpp"
```

Include dependency graph for main.c:



### Functions

- void [SystemClock\\_Config](#) (void)  
*System Clock Configuration.*
- void [StartDefaultTask](#) (void const \*argument)  
*Function implementing the defaultTask thread.*
- int [main](#) (void)  
*The application entry point.*
- void [HAL\\_TIM\\_PeriodElapsedCallback](#) (TIM\_HandleTypeDef \*htim)  
*Period elapsed callback in non blocking mode.*
- void [Error\\_Handler](#) (void)  
*This function is executed in case of error occurrence.*
- void [assert\\_failed](#) (uint8\_t \*file, uint32\_t line)  
*Reports the name of the source file and the source line number where the assert\_param error has occurred.*

### Variables

- DMA\_HandleTypeDef [hdma\\_spi1\\_rx](#)

#### 8.28.1 Detailed Description

### Attention

© Copyright (c) 2020 STMicroelectronics. All rights reserved.

This software component is licensed by ST under BSD 3-Clause license, the "License"; You may not use this file except in compliance with the License. You may obtain a copy of the License at: [opensource.org/licenses/BSD-3-Clause](https://opensource.org/licenses/BSD-3-Clause)

## 8.28.2 Function Documentation

### 8.28.2.1 void assert\_failed ( uint8\_t \* file, uint32\_t line )

#### Parameters

<i>file</i>	pointer to the source file name
<i>line</i>	assert_param error line source number

#### Return values

<i>None</i>	
-------------	--

### 8.28.2.2 void Error\_Handler ( void )

#### Return values

<i>None</i>	
-------------	--

### 8.28.2.3 void HAL\_TIM\_PeriodElapsedCallback ( TIM\_HandleTypeDef \* htim )

#### Note

This function is called when TIM14 interrupt took place, inside HAL\_TIM\_IRQHandler(). It makes a direct call to HAL\_IncTick() to increment a global variable "uwTick" used as application time base.

#### Parameters

<i>htim</i>	: TIM handle
-------------	--------------

#### Return values

<i>None</i>	
-------------	--

### 8.28.2.4 int main ( void )

#### Return values

<i>int</i>	
------------	--

#### 8.28.2.5 void StartDefaultTask ( void const \* *argument* )

##### Parameters

<i>argument</i>	Not used
-----------------	----------

##### Return values

<i>None</i>	
-------------	--

#### 8.28.2.6 void SystemClock\_Config ( void )

##### Return values

<i>None</i>	
-------------	--

Configure the main internal regulator output voltage

Initializes the CPU, AHB and APB busses clocks

Activate the Over-Drive mode

Initializes the CPU, AHB and APB busses clocks

### 8.28.3 Variable Documentation

#### 8.28.3.1 DMA\_HandleTypeDef hdma\_spi1\_rx

File Name : stm32f4xx\_hal\_msp.c Description : This file provides code for the MSP Initialization and de-Initialization codes.

##### Attention

© Copyright (c) 2020 STMicroelectronics. All rights reserved.

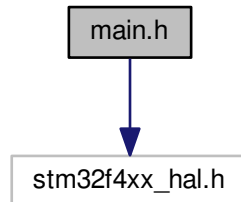
This software component is licensed by ST under BSD 3-Clause license, the "License"; You may not use this file except in compliance with the License. You may obtain a copy of the License at: [opensource.org/licenses/BSD-3-Clause](https://opensource.org/licenses/BSD-3-Clause)

## 8.29 main.h File Reference

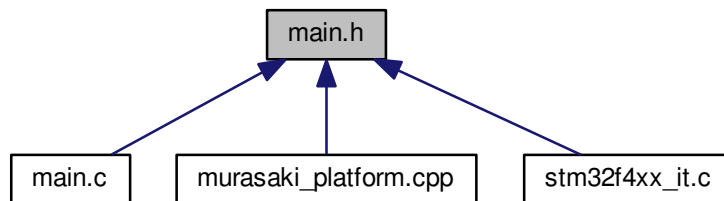
: Header for [main.c](#) file. This file contains the common defines of the application.

```
#include "stm32f4xx_hal.h"
```

Include dependency graph for main.h:



This graph shows which files directly or indirectly include this file:



### Functions

- void [Error\\_Handler](#) (void)

*This function is executed in case of error occurrence.*

#### 8.29.1 Detailed Description

#### Attention

© Copyright (c) 2020 STMicroelectronics. All rights reserved.

This software component is licensed by ST under BSD 3-Clause license, the "License"; You may not use this file except in compliance with the License. You may obtain a copy of the License at: [opensource.org/licenses/BSD-3-Clause](https://opensource.org/licenses/BSD-3-Clause)

## 8.29.2 Function Documentation

## 8.29.2.1 void Error\_Handler ( void )

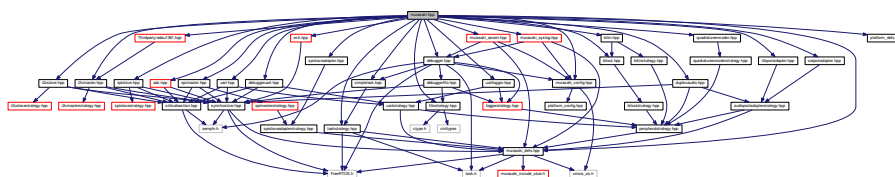
Return values

None	
------	--

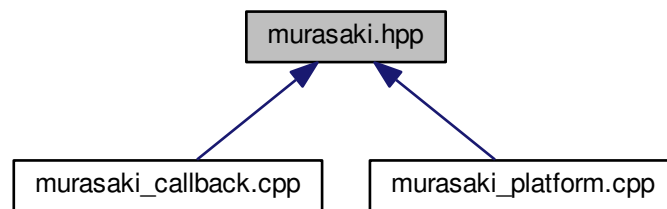
## 8.30 murasaki.hpp File Reference

Application include file for Murasaki class library.

```
#include <debugger.hpp>
#include <fifostrategy.hpp>
#include <taskstrategy.hpp>
#include "murasaki_config.hpp"
#include "murasaki_defs.hpp"
#include "simpletask.hpp"
#include "duplexaudio.hpp"
#include "uart.hpp"
#include "debuggeruart.hpp"
#include "spimaster.hpp"
#include "spislave.hpp"
#include "spislaveadapter.hpp"
#include "i2cmaster.hpp"
#include "i2cslave.hpp"
#include "bitin.hpp"
#include "bitout.hpp"
#include "saiportheadapter.hpp"
#include "i2sportadapter.hpp"
#include "quadratureencoder.hpp"
#include "adc.hpp"
#include "exti.hpp"
#include "murasaki_utility.hpp"
#include "Thirdparty/adau1361.hpp"
#include "uartlogger.hpp"
#include "murasaki_assert.hpp"
#include "murasaki_syslog.hpp"
#include "platform_defs.hpp"
Include dependency graph for murasaki.hpp:
```



This graph shows which files directly or indirectly include this file:



#### 8.30.1 Detailed Description

##### Date

2018/01/21

##### Author

Seiichi "Suikan" Horie

Application can include only this file. Other essential header files are automatically included from this file.

### 8.31 murasaki\_0\_intro.hpp File Reference

Doxygen document file. No need to include.

#### 8.31.1 Detailed Description

##### Date

2020/03/19

##### Author

Seiichi "Suikan" Horie

### 8.32 murasaki\_1\_env.hpp File Reference

Doxygen document file. No need to include.

**8.32.1 Detailed Description****Date**

2020/03/19

**Author**

Seiichi "Suikan" Horie

**8.33 murasaki\_2\_ug.hpp File Reference**

Doxygen document file. No need to include.

**8.33.1 Detailed Description****Date**

2018/02/01

**Author**

Seiichi "Suikan" Horie

**8.34 murasaki\_3\_pg.hpp File Reference**

Porting Guide.

**8.34.1 Detailed Description****Date**

May 25, 2018

**Author**

Seiichi "Suikan" Horie

**8.35 murasaki\_4\_mod.hpp File Reference**

Module definition.

**8.35.1 Detailed Description****Date**

May 25, 2018

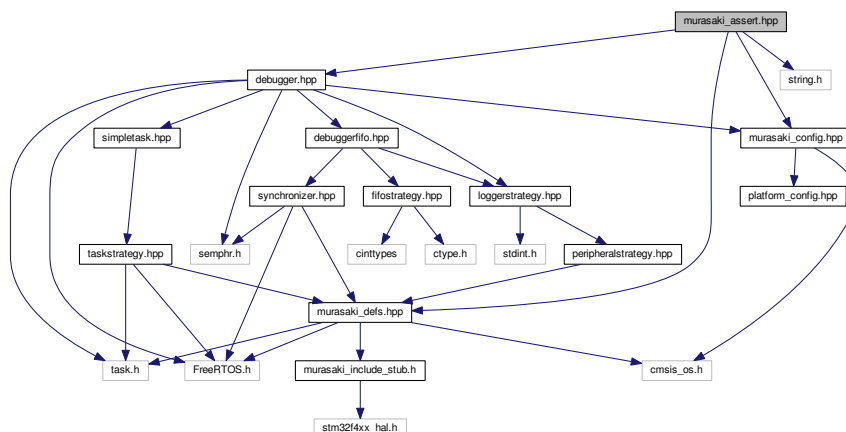
**Author**

Seiichi "Suikan" Horie

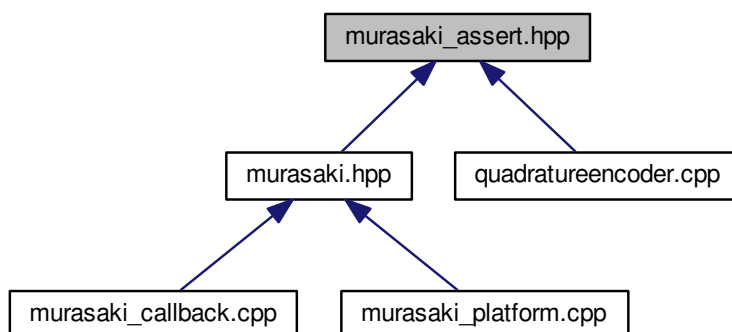
### 8.36 murasaki\_assert.hpp File Reference

Assertion definition.

```
#include <debugger.hpp>
#include "murasaki_config.hpp"
#include "murasaki_defs.hpp"
#include <string.h>
Include dependency graph for murasaki_assert.hpp:
```



This graph shows which files directly or indirectly include this file:



#### Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.



## Macros

- #define [MURASAKI\\_ASSERT](#)(COND)  
*Assert the COND is true.*
- #define [MURASAKI\\_PRINT\\_ERROR](#)(ERR)  
*Print ERR if ERR is true.*

### 8.36.1 Detailed Description

#### Date

2018/01/31

#### Author

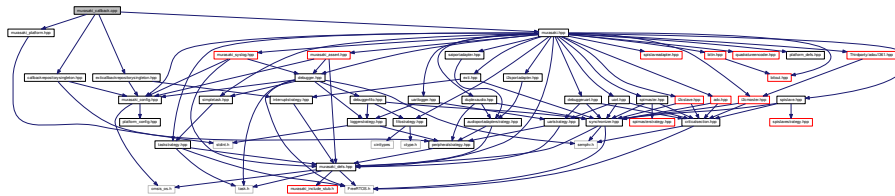
Seiichi "Suikan" Horie

## 8.37 murasaki\_callback.cpp File Reference

A glue file between the callback and HAL/RTOS.

```
#include <murasaki_platform.hpp>
#include "callbackrepositorysingleton.hpp"
#include "extcallbackrepositorysingleton.hpp"
#include "murasaki.hpp"
```

Include dependency graph for murasaki\_callback.cpp:



## Functions

- void [HAL\\_UART\\_TxCpltCallback](#) (UART\_HandleTypeDef \*huart)  
*Essential to sync up with UART.*
- void [HAL\\_UART\\_RxCpltCallback](#) (UART\_HandleTypeDef \*huart)  
*Essential to sync up with UART.*
- void [HAL\\_UART\\_ErrorCallback](#) (UART\_HandleTypeDef \*huart)  
*Optional error handling of UART.*
- void [HAL\\_SPI\\_TxRxCpltCallback](#) (SPI\_HandleTypeDef \*hspl)  
*Essential to sync up with SPI.*
- void [HAL\\_SPI\\_ErrorCallback](#) (SPI\_HandleTypeDef \*hspl)  
*Optional error handling of SPI.*
- void [HAL\\_I2C\\_MasterTxCpltCallback](#) (I2C\_HandleTypeDef \*hi2c)  
*Essential to sync up with I2C.*
- void [HAL\\_I2C\\_MasterRxCpltCallback](#) (I2C\_HandleTypeDef \*hi2c)

- *Essential to sync up with I2C.*  
void [HAL\\_I2C\\_SlaveTxCpltCallback](#) (I2C\_HandleTypeDef \*hi2c)
- *Essential to sync up with I2C.*  
void [HAL\\_I2C\\_SlaveRxCpltCallback](#) (I2C\_HandleTypeDef \*hi2c)
- *Essential to sync up with I2C.*  
void [HAL\\_I2C\\_ErrorCallback](#) (I2C\_HandleTypeDef \*hi2c)
- *Optional error handling of I2C.*  
void [HAL\\_ADC\\_ConvCpltCallback](#) (ADC\_HandleTypeDef \*hadc)
- *Optional interrupt handler for ADC.*  
void [HAL\\_ADC\\_ErrorCallback](#) (ADC\_HandleTypeDef \*hadc)
- *Optional interrupt handler for ADC.*  
void [HAL\\_SAI\\_RxHalfCpltCallback](#) (SAI\_HandleTypeDef \*hsai)
- *Optional SAI interrupt handler at buffer transfer halfway.*  
void [HAL\\_SAI\\_RxCpltCallback](#) (SAI\_HandleTypeDef \*hsai)
- *Optional SAI interrupt handler at buffer transfer complete.*  
void [HAL\\_SAI\\_ErrorCallback](#) (SAI\_HandleTypeDef \*hsai)
- *Optional SAI error interrupt handler.*  
void [HAL\\_I2S\\_RxHalfCpltCallback](#) (I2S\_HandleTypeDef \*hi2s)
- *Optional I2S interrupt handler at buffer transfer halfway.*  
void [HAL\\_I2S\\_RxCpltCallback](#) (I2S\_HandleTypeDef \*hi2s)
- *Optional I2S interrupt handler at buffer transfer complete.*  
void [HAL\\_I2S\\_ErrorCallback](#) (I2S\_HandleTypeDef \*hi2s)
- *Optional I2S error interrupt handler.*  
void [HAL\\_GPIO\\_EXTI\\_Callback](#) (uint16\_t GPIO\_Pin)
- *Optional interrupt handling of EXTI.*  
void [CustomAssertFailed](#) (uint8\_t \*file, uint32\_t line)
- *Hook for the assert\_failure() in main.c.*  
void [PrintFaultResult](#) (unsigned int \*stack\_pointer)
- *Printing out the context information.*

### 8.37.1 Detailed Description

#### Date

2020/04/12

#### Author

Seiichi "Suikan" Horie

### 8.37.2 Function Documentation

#### 8.37.2.1 void HAL\_ADC\_ConvCpltCallback ( ADC\_HandleTypeDef \* hadc )

##### Parameters

<i>hadc</i>	Pointer to the Adc control structure.
-------------	---------------------------------------

This is called from inside of HAL when an ADC conversion is done.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default error interrupt call back.

#### 8.37.2.2 void HAL\_ADC\_ErrorCallback ( ADC\_HandleTypeDef \* *hadc* )

##### Parameters

<i>hadc</i>	Pointer to the Adc control structure.
-------------	---------------------------------------

This is called from inside of HAL when an ADC conversion failed by Error.

#### 8.37.2.3 void HAL\_I2C\_MasterRxCpltCallback ( I2C\_HandleTypeDef \* *hi2c* )

##### Parameters

<i>hi2c</i>	
-------------	--

This is called from inside of HAL when an I2C receive done interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default RX interrupt call back.

In this call back, the uart device handle have to be passed to the [murasaki::Uart::ReceiveCompleteCallback\(\)](#) function.

#### 8.37.2.4 void HAL\_I2C\_SlaveRxCpltCallback ( I2C\_HandleTypeDef \* *hi2c* )

##### Parameters

<i>hi2c</i>	
-------------	--

This is called from inside of HAL when an I2C receive done interrupt is accepted.

STM32Cube HAL has same name function internally. That function is invoked whenever an relevant interrupt happens. In the other hand, that function is declared as weak bound. As a result, this function overrides the default RX interrupt call back.

In this call back, the I2C slave device handle have to be passed to the [murasaki::I2cSlave::ReceiveCompleteCallback\(\)](#) function.

#### 8.37.2.5 void PrintFaultResult ( unsigned int \* *stack\_pointer* )

##### Parameters

<i>stack_pointer</i>	retrieved stack pointer before interrupt / exception.
----------------------	---

Do not call from application. This is murasaki\_internal\_only.

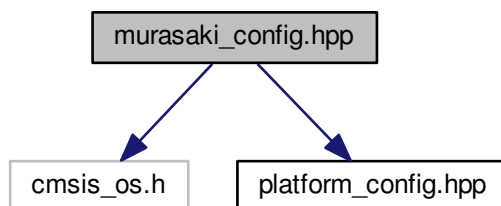
### 8.38 murasaki\_config.hpp File Reference

Configuration file for platform.

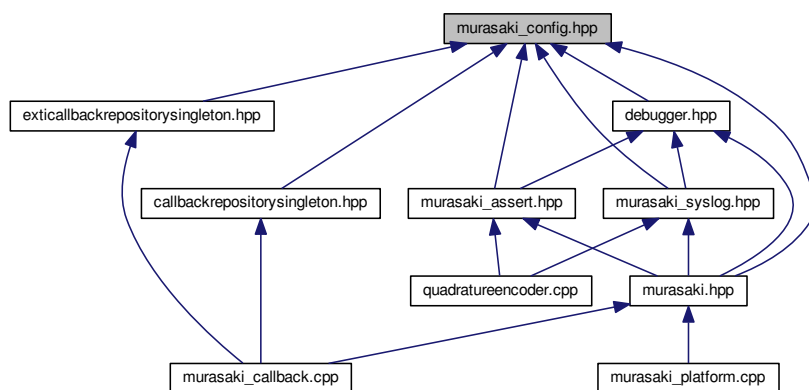
```
#include <cmsis_os.h>
```

```
#include <platform_config.hpp>
```

Include dependency graph for murasaki\_config.hpp:



This graph shows which files directly or indirectly include this file:



#### Macros

- `#define PLATFORM_CONFIG_DEBUG_LINE_SIZE 256`  
*Size of one line[byte] in the debug printf.*
- `#define PLATFORM_CONFIG_DEBUG_BUFFER_SIZE 4096`  
*Size[byte] of the circular buffer to be transmitted through the serial port.*
- `#define PLATFORM_CONFIG_DEBUG_SERIAL_TIMEOUT (murasaki::kwmsIndefinitely)`  
*Timeout of the serial port to transmit the string through the Debug class.*
- `#define PLATFORM_CONFIG_DEBUG_TASK_STACK_SIZE 256`  
*Size[Byte] of the task inside Debug class.*
- `#define PLATFORM_CONFIG_DEBUG_TASK_PRIORITY murasaki::ktpHigh`

- The task priority of the debug task.  
• #define `MURASAKI_CONFIG_NODEBUG` false  
Suppress `MURASAKI_ASSERT` macro.
- #define `MURASAKI_CONFIG_NOCYCCNT` false  
Doesn't run the `CYCCNT` counter.
- #define `NUM_OF_CALLBACK_OBJECTS` 8  
The number of the interrupt callback handling objects.
- #define `NUM_OF_EXTI_OBJECTS` 8  
The number of the `EXTI` callback handling objects.

### 8.38.1 Detailed Description

#### Date

2018/01/03

#### Author

Seiichi "Suikan" Horie

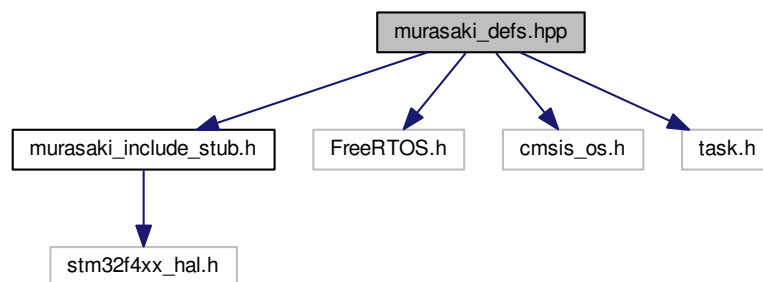
To override the configuration, define the same name macro inside `application_config.hpp`

## 8.39 murasaki\_defs.hpp File Reference

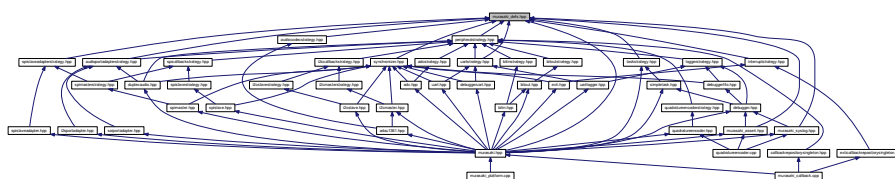
common definition of the platform.

```
#include "murasaki_include_stub.h"
#include <FreeRTOS.h>
#include <cmsis_os.h>
#include <task.h>
```

Include dependency graph for `murasaki_defs.hpp`:



This graph shows which files directly or indirectly include this file:



## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

## Enumerations

## Functions

- void [murasaki::InitCycleCounter](#) ()  
*Initialize and start the cycle counter.*
- unsigned int [murasaki::GetCycleCounter](#) ()  
*Obtain the current cycle count of CYCCNT register.*

## 8.39.1 Detailed Description

## Date

2017/11/05

## Author

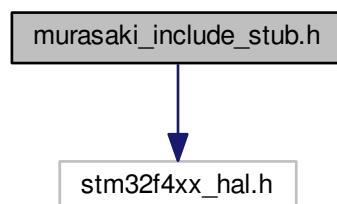
Seiichi "Suikan" Horie

8.40 [murasaki\\_include\\_stub.h](#) File Reference

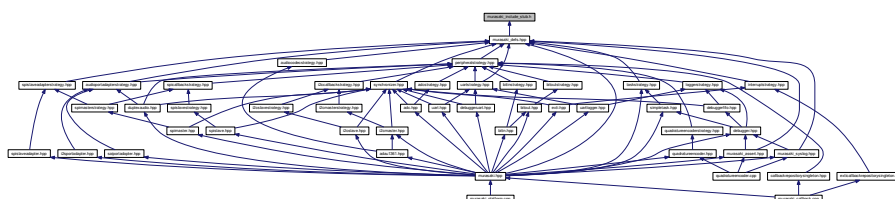
Stub to include the HAL headers.

```
#include <stm32f4xx_hal.h>
```

Include dependency graph for [murasaki\\_include\\_stub.h](#):



This graph shows which files directly or indirectly include this file:



### 8.40.1 Detailed Description

The CubeIDE add the STM32 microprocessor product name as pre-defined macro when a file is compiled. For example, following is the macro definition for STM32F446 processor at the compiler command line.

-DSTM32F446xx

On the other hand, this is not enough to determine the appropriate include file inside `murasaki_defs.hpp`. As a result, there are difficulties to include the appropriate file.

One of the naive approach is to enumerate all possible pre-defined macro to determine the filename as following.

```
#elif defined (STM32F405xx) || defined (STM32F415xx) || defined (STM32F407xx) || defined (STM32F417xx) || *
    defined (STM32F427xx) || defined (STM32F437xx) || defined (STM32F429xx) || defined (STM32F439xx) || *
    defined (STM32F401xC) || defined (STM32F401xE) || defined (STM32F410Tx) || defined (STM32F410Cx) || *
    defined (STM32F410Rx) || defined (STM32F411xE) || defined (STM32F446xx) || defined (STM32F469xx) || *
    defined (STM32F479xx) || defined (STM32F412Cx) || defined (STM32F412Rx) || defined (STM32F412Vx) || *    defined
    (STM32F412Zx) || defined (STM32F413xx) || defined (STM32F423xx)
#include "stm32f4xx_hal.h"
```

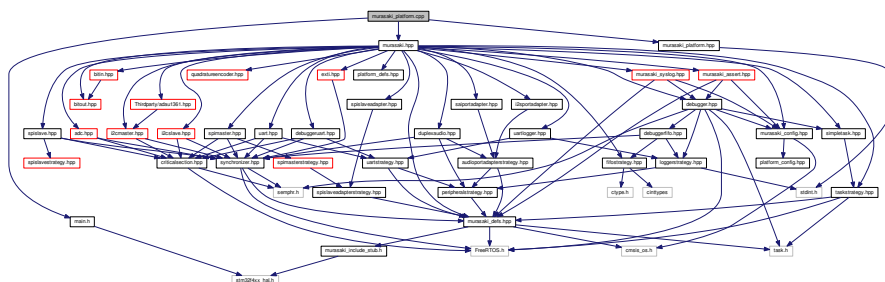
This is easy to understand. But boring to maintain.

This stub is alternate way. [murasaki\\_defs.hpp](#) is including this file ([murasaki\\_include\\_stub.h](#)). And this stub file include the appropriate HAL header file. This stub file is generated by `murasaki/install` script. Thus, user doesn't need to maintain this file.

## 8.41 murasaki\_platform.cpp File Reference

A glue file between the user application and HAL/RTOS.

```
#include <murasaki_platform.hpp>
#include "main.h"
#include "murasaki.hpp"
Include dependency graph for murasaki_platform.cpp:
```



## Functions

- void `InitPlatform ()`  
*Initialize the platform variables.*
- void `ExecPlatform ()`  
*The body of the real application.*

### 8.41.1 Detailed Description

#### Date

2018/05/20

#### Author

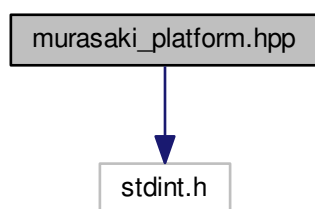
Seiichi "Suikan" Horie

## 8.42 murasaki\_platform.hpp File Reference

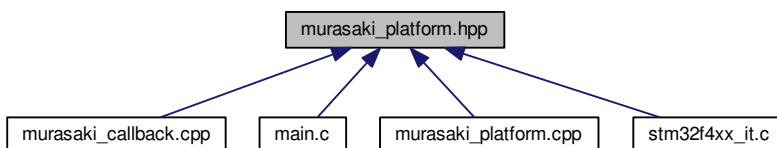
An interface for the applicaiton from murasaki library to [main.c](#).

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

Include dependency graph for murasaki\_platform.hpp:



This graph shows which files directly or indirectly include this file:



#### Functions

- void [InitPlatform](#) ()  
*Initialize the platform variables.*
- void [ExecPlatform](#) ()  
*The body of the real application.*
- void [CustomAssertFailed](#) (uint8\_t \*file, uint32\_t line)  
*Hook for the assert\_failure() in [main.c](#).*



- void [CustomDefaultHandler](#) ()  
*Hook for the default exception handler. Never return.*
- void [PrintFaultResult](#) (unsigned int \*stack\_pointer)  
*Printing out the context information.*
- void [MasterTaskBodyFunction](#) (const void \*ptr)  
*Master test task.*
- void [SlaveTaskBodyFunction](#) (const void \*ptr)  
*Demonstration task.*

### 8.42.1 Detailed Description

#### Date

2017/11/12

#### Author

Seiichi "Suikan" Horie

The resources below are implemented in the [murasaki\\_platform.cpp](#) and serve as glue to the [main.c](#).

### 8.42.2 Function Documentation

#### 8.42.2.1 void MasterTaskBodyFunction ( const void \* ptr )

##### Parameters

<i>ptr</i>	Pointer to the parameter block
------------	--------------------------------

Task body function of the test. Call test subprogram step by step.

#### 8.42.2.2 void PrintFaultResult ( unsigned int \* stack\_pointer )

##### Parameters

<i>stack_pointer</i>	retrieved stack pointer before interrupt / exception.
----------------------	---

Do not call from application. This is `murasaki_internal_only`.

#### 8.42.2.3 void SlaveTaskBodyFunction ( const void \* ptr )

##### Parameters

<i>ptr</i>	Pointer to the parameter block
------------	--------------------------------

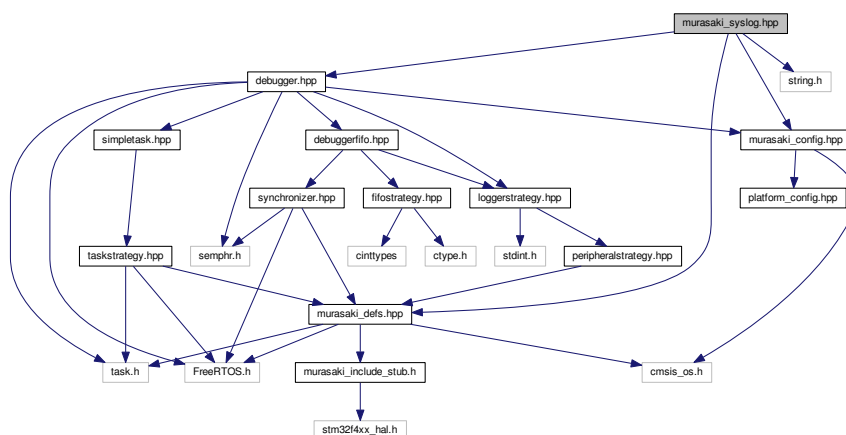
Task body function as demonstration of the [murasaki::SimpleTask](#).

You can delete this function if you don't use.

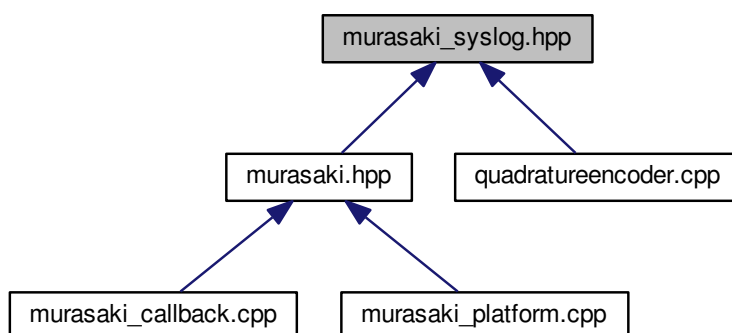
### 8.43 murasaki\_syslog.hpp File Reference

Syslog definition.

```
#include <debugger.hpp>
#include "murasaki_config.hpp"
#include "murasaki_defs.hpp"
#include "string.h"
Include dependency graph for murasaki_syslog.hpp:
```



This graph shows which files directly or indirectly include this file:



#### Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

## Macros

- `#define MURASAKI_SYSLOG(OBJPTR, FACILITY, SEVERITY, FORMAT, ...)`  
*output The debug message*

## Functions

- void `murasaki::SetSyslogSererityThreshold` (`murasaki::SyslogSeverity` severity)  
*Set the syslog severity threshold.*
- void `murasaki::SetSyslogFacilityMask` (`uint32_t` mask)  
*Set the syslog facility mask.*
- void `murasaki::AddSyslogFacilityToMask` (`murasaki::SyslogFacility` facility)  
*Add Syslog facility to the filter mask.*
- void `murasaki::RemoveSyslogFacilityFromMask` (`murasaki::SyslogFacility` facility)  
*Remove Syslog facility to the filter mask.*
- bool `murasaki::AllowedSyslogOut` (`murasaki::SyslogFacility` facility, `murasaki::SyslogSeverity` severity)  
*Check if given facility and severity message is allowed to output.*

### 8.43.1 Detailed Description

#### Date

2018/09/01

#### Author

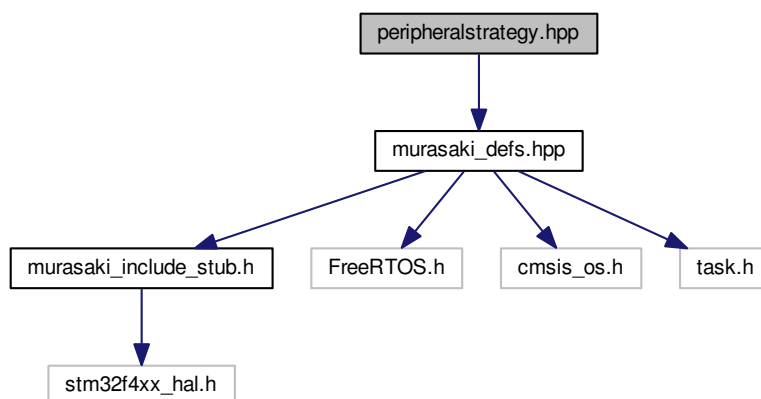
Seiichi "Suikan" Horie

## 8.44 peripheralstrategy.hpp File Reference

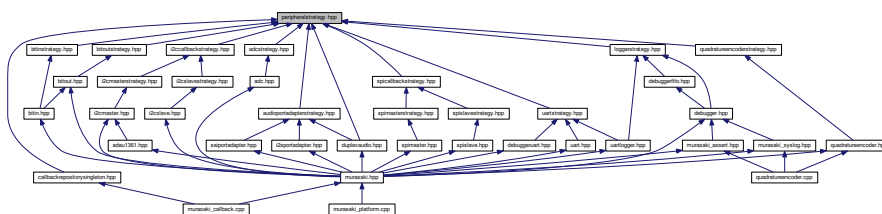
Mother of All peripheral.

```
#include "murasaki_defs.hpp"
```

Include dependency graph for peripheralstrategy.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::PeripheralStrategy](#)  
*Mother of all peripheral class.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.44.1 Detailed Description

#### Date

2018/04/26

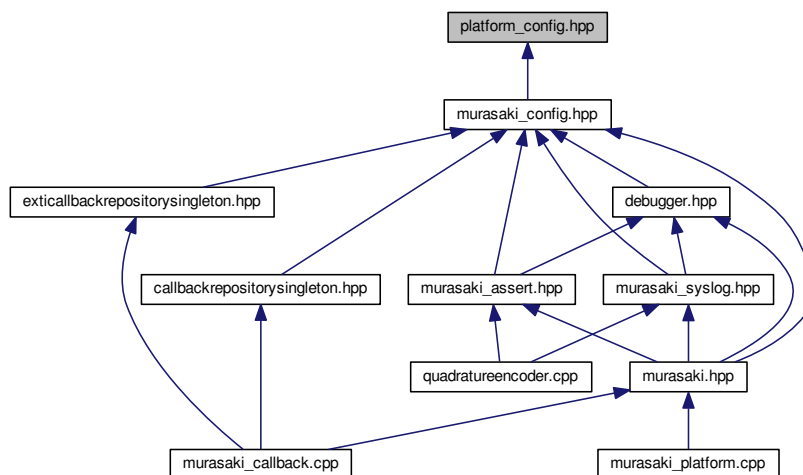
#### Author

: Seiichi "Suikan" Horie

## 8.45 platform\_config.hpp File Reference

Application dependent configuration.

This graph shows which files directly or indirectly include this file:



## Macros

- `#define MURASAKI_CONFIG_NOSYSLOG false`  
*Suppress `MURASAKI_SYSLOG` macro.*

## 8.45.1 Detailed Description

## Date

2018/01/07

## Author

Seiichi "Suikan" Horie

If you want to override the macro definition inside `platform_config.hpp`, add your definition here.

## 8.45.2 Macro Definition Documentation

8.45.2.1 `#define MURASAKI_CONFIG_NOSYSLOG false`

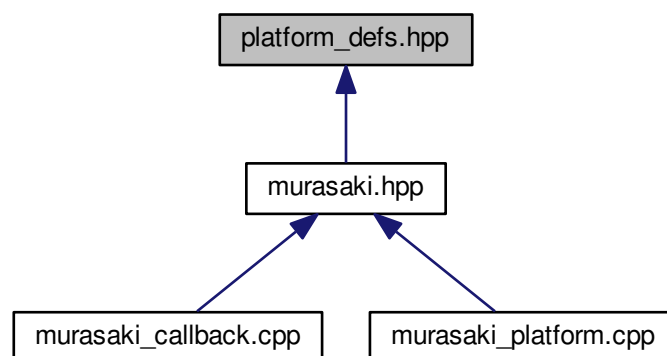
Set this macro to true, to discard the `MURASAKI_SYSLOG`. Set this macro false, to use the syslog.

To override the definition here, define same macro inside `platform_config.hpp`.

## 8.46 platform\_defs.hpp File Reference

Murasaki platform customize file.

This graph shows which files directly or indirectly include this file:





## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

## 8.47.1 Detailed Description

## Date

2020/02/19

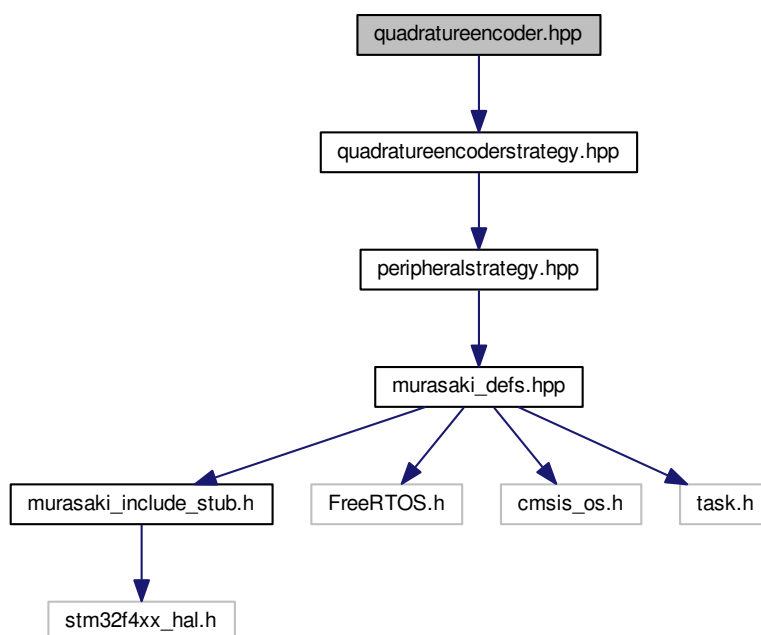
## Author

takemasa

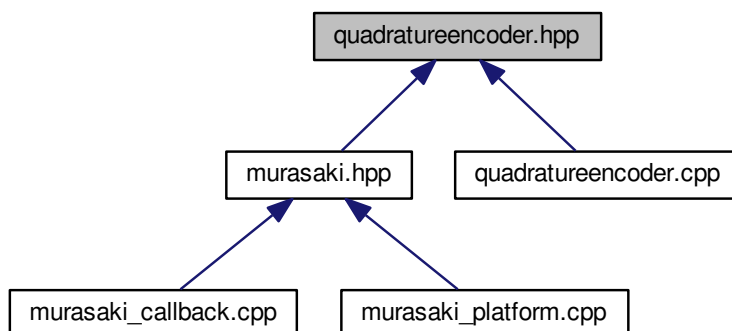
## 8.48 quadratureencoder.hpp File Reference

```
#include <quadratureencoderstrategy.hpp>
```

Include dependency graph for quadratureencoder.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::QuadratureEncoder](#)  
*Quadrature Encoder class.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.48.1 Detailed Description

##### Date

2020/02/19

##### Author

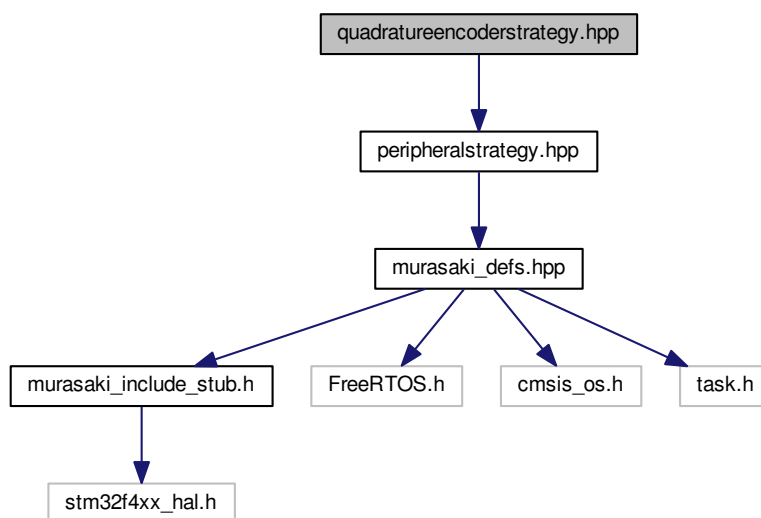
takemasa

#### 8.49 quadratureencoderstrategy.hpp File Reference

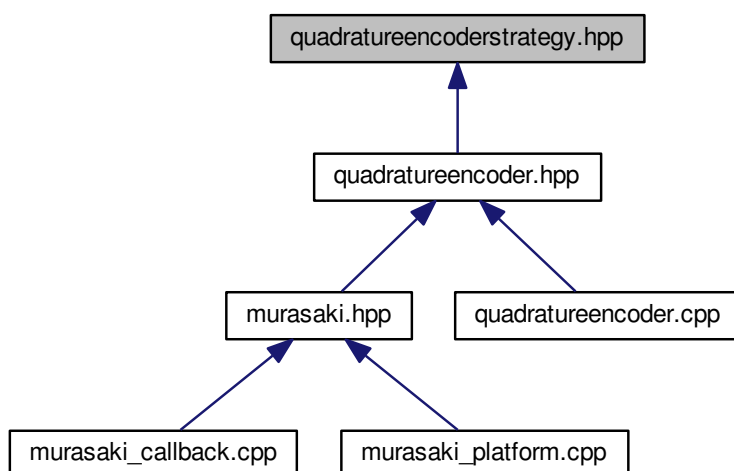
```
#include <peripheralstrategy.hpp>
```



Include dependency graph for quadratureencoderstrategy.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class `murasaki::QuadratureEncoderStrategy`  
Strategy class for the quadrature encoder.

## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

## 8.49.1 Detailed Description

## Date

2020/02/19

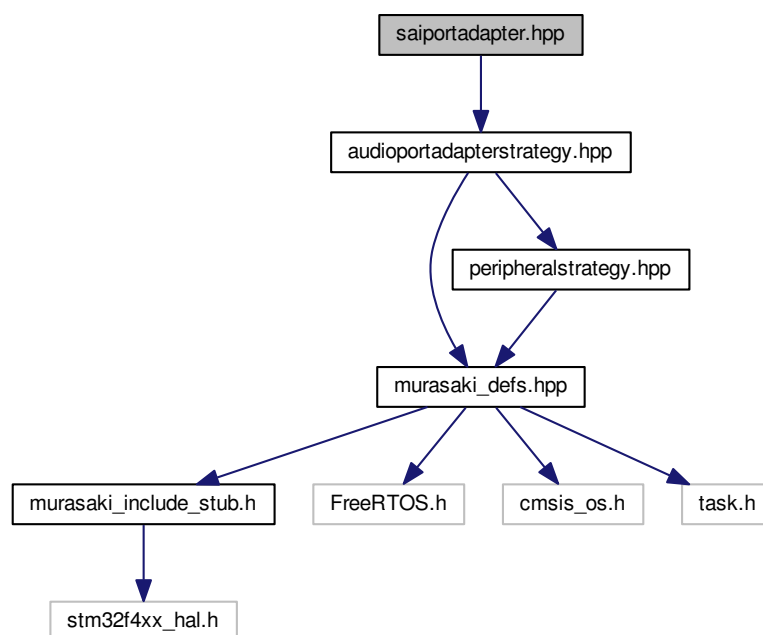
## Author

Seiichi Horie

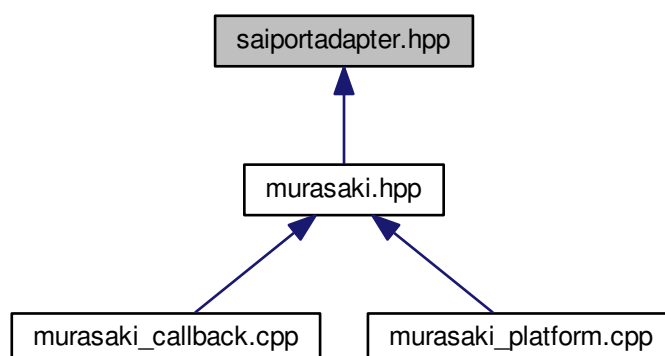
## 8.50 saiportadapter.hpp File Reference

```
#include <audioportadapterstrategy.hpp>
```

Include dependency graph for saiportadapter.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::SaiPortAdapter](#)  
*Adapter as SAI audio port.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.50.1 Detailed Description

##### Date

2019/07/28

##### Author

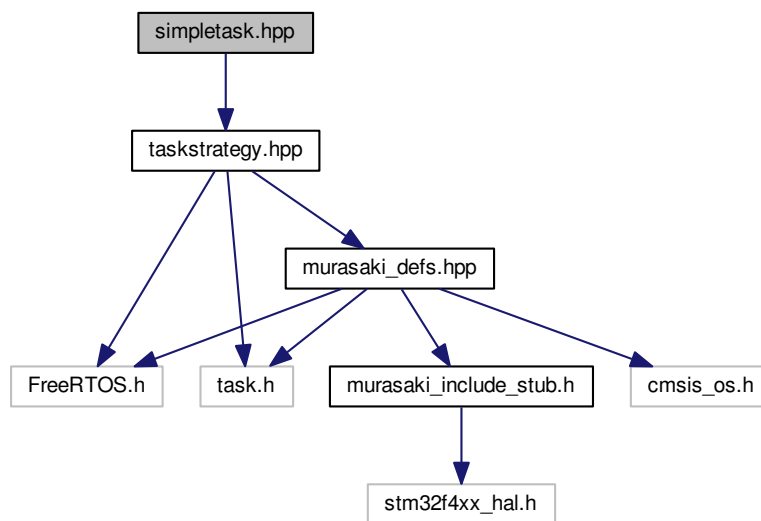
takemasa

## 8.51 simpletask.hpp File Reference

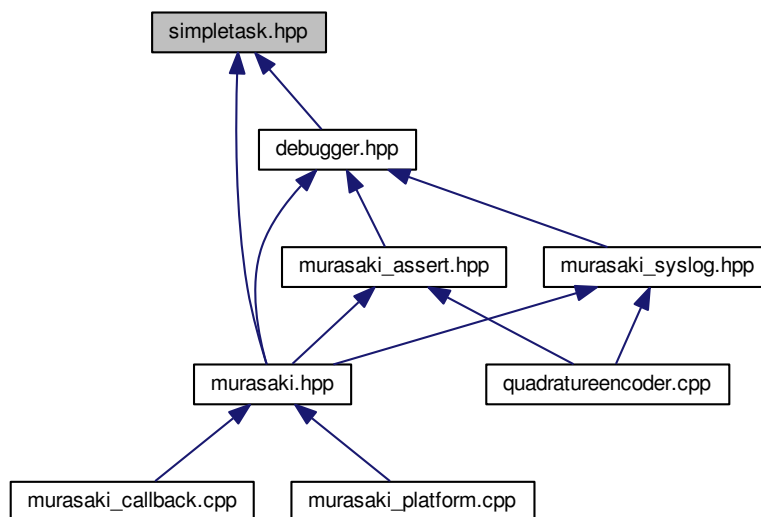
Simplified Task class.

```
#include <taskstrategy.hpp>
```

Include dependency graph for simpletask.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class `murasaki::SimpleTask`  
An easy to use task class.

## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

## 8.51.1 Detailed Description

## Date

2019/02/03

## Author

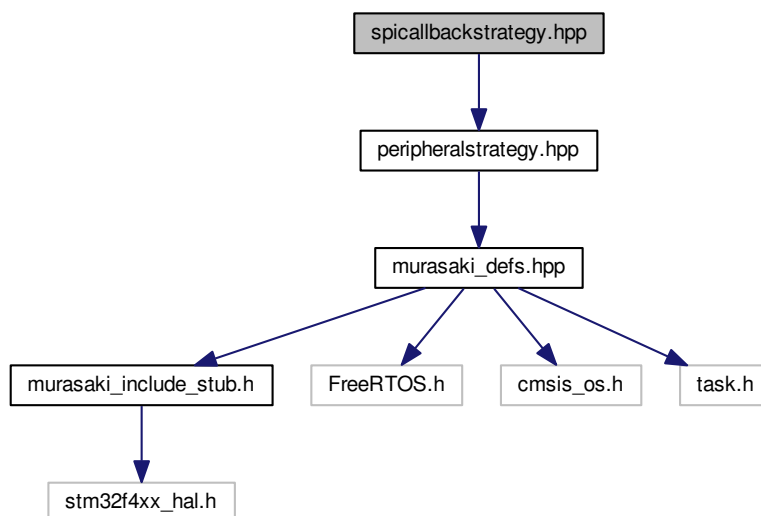
Seiichi "Suikan" Horie

## 8.52 spicallbackstrategy.hpp File Reference

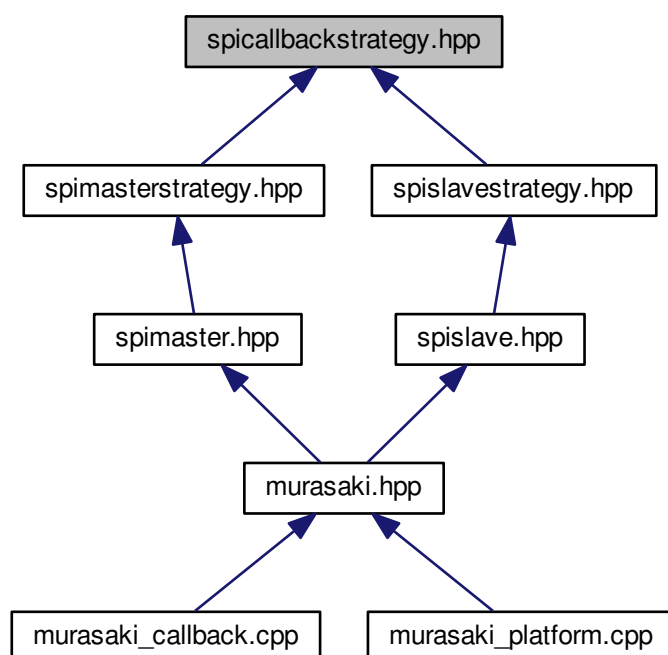
SPI root class.

```
#include <peripheralstrategy.hpp>
```

Include dependency graph for spicallbackstrategy.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::SpiCallbackStrategy](#)  
*Root class of the SPI.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.52.1 Detailed Description

#### Date

2020/04/11

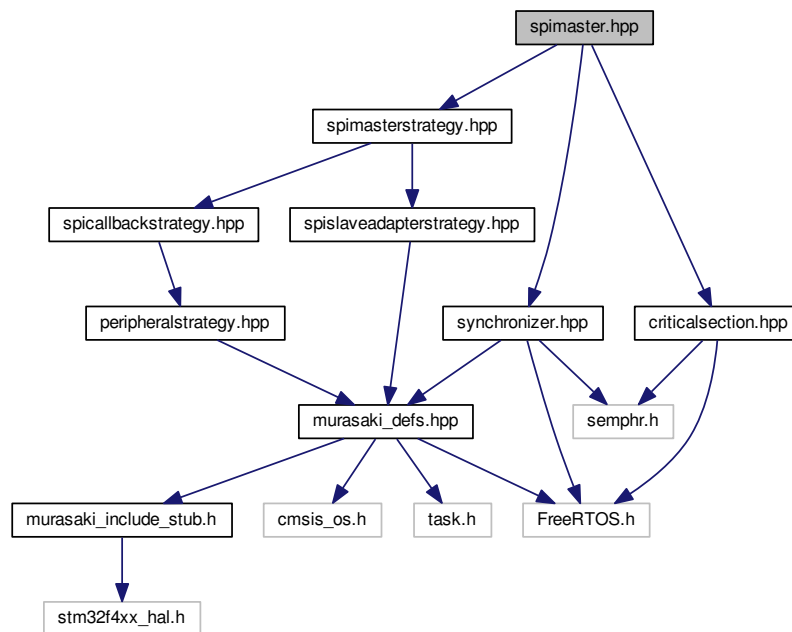
#### Author

: Seiichi "Suikan" Horie

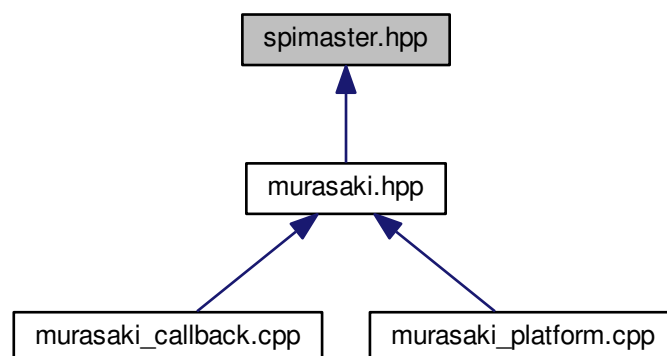
## 8.53 spimaster.hpp File Reference

SPI Master. Thread safe and synchronous IO.

```
#include <spimasterstrategy.hpp>
#include <synchronizer.hpp>
#include "criticalsection.hpp"
Include dependency graph for spimaster.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::SpiMaster](#)

*Thread safe, synchronous and blocking IO. Encapsulating SPI master. Based on STM32Cube HAL driver and FreeRTOS.*

## Namespaces

- [murasaki](#)

*Personal Platform parts collection.*

## 8.53.1 Detailed Description

## Date

2018/02/14

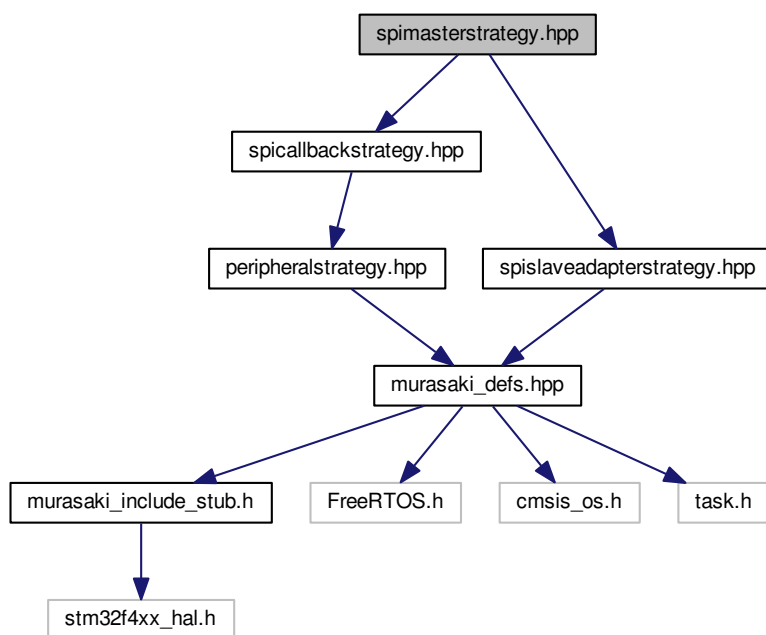
## Author

Seiichi "Suikan" Horie

## 8.54 spimasterstrategy.hpp File Reference

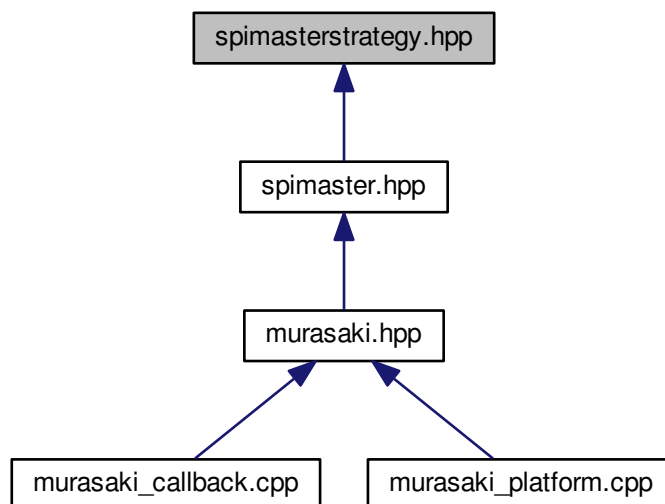
SPI master root class.

```
#include <spicallbackstrategy.hpp>
#include <spislaveadapterstrategy.hpp>
Include dependency graph for spimasterstrategy.hpp:
```





This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::SpiMasterStrategy](#)  
*Root class of the SPI master.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.54.1 Detailed Description

##### Date

2018/02/11

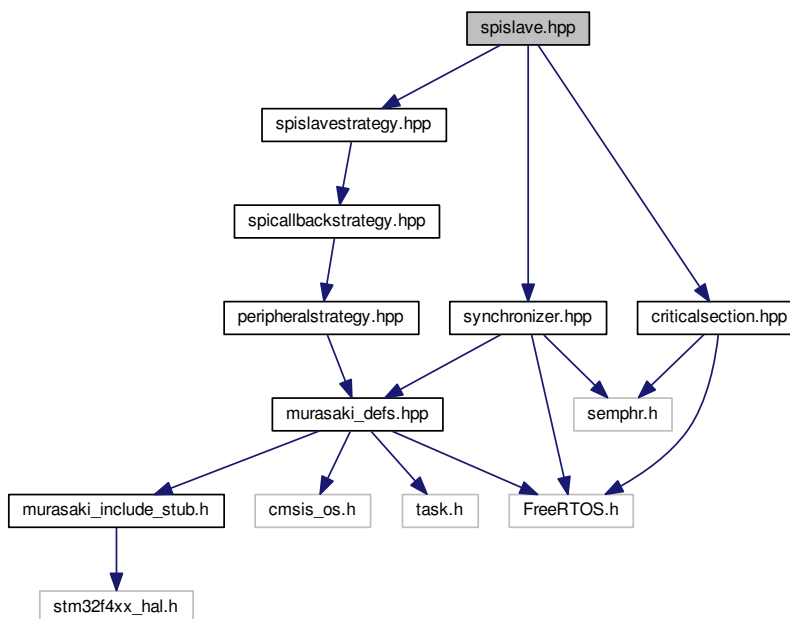
##### Author

: Seiichi "Suikan" Horie

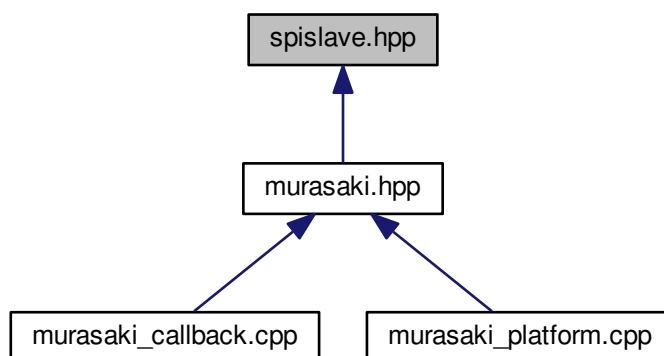
## 8.55 spislave.hpp File Reference

SPI Slave. Thread safe and synchronous IO.

```
#include <spislavestrategy.hpp>
#include <synchronizer.hpp>
#include "criticalsection.hpp"
Include dependency graph for spislave.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::SpiSlave](#)

*Thread safe, synchronous and blocking IO. Encapsulating SPI slave. Based on STM32Cube HAL driver and FreeRTOS.*

## Namespaces

- [murasaki](#)

*Personal Platform parts collection.*

## 8.55.1 Detailed Description

## Date

2018/02/14

## Author

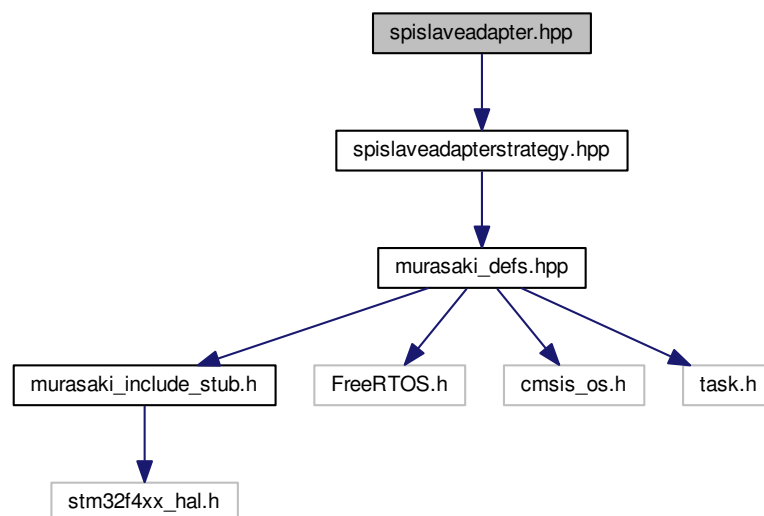
Seiichi "Suikan" Horie

## 8.56 spislaveadapter.hpp File Reference

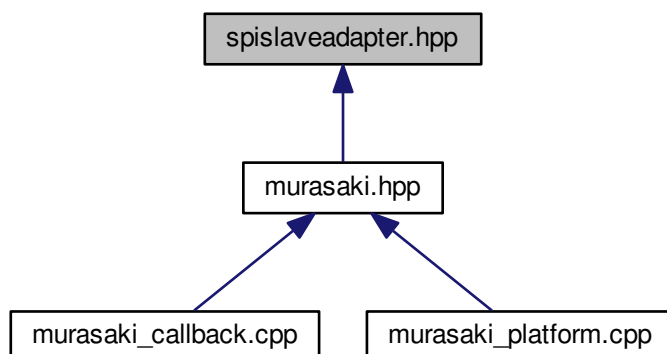
STM32 SPI slave speifire.

```
#include <spislaveadapterstrategy.hpp>
```

Include dependency graph for spislaveadapter.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::SpiSlaveAdapter](#)  
*A specifier of SPI slave.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.56.1 Detailed Description

##### Date

2018/02/17

##### Author

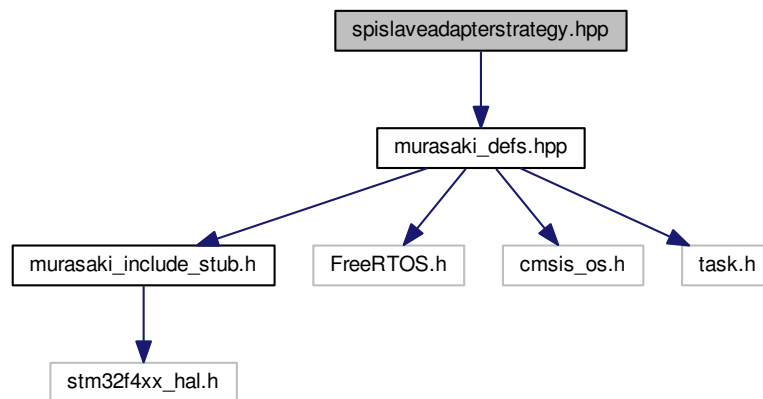
Seiichi "Suikan" Horie

#### 8.57 spislaveadapterstrategy.hpp File Reference

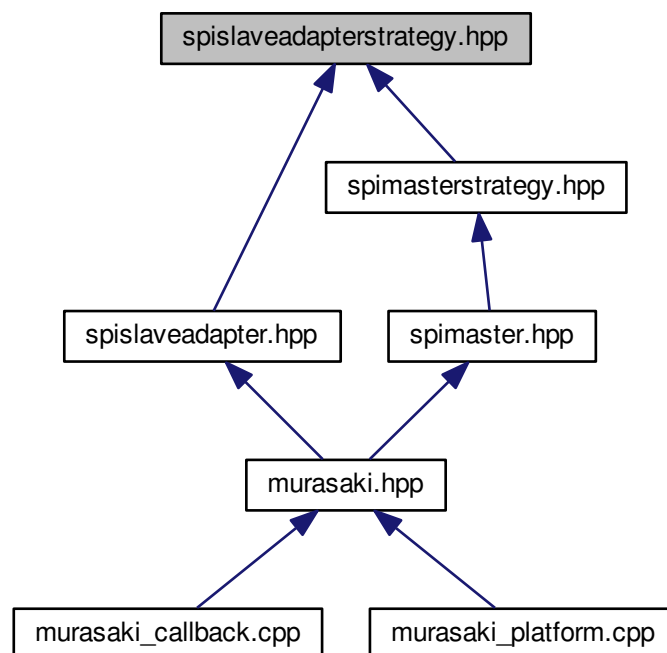
Abstract class of SPI slave specification.

```
#include "murasaki_defs.hpp"
```

Include dependency graph for spislaveadapterstrategy.hpp:



This graph shows which files directly or indirectly include this file:



#### Classes

- class `murasaki::SpiSlaveAdapterStrategy`  
Definition of the root class of SPI slave adapter.

## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

### 8.57.1 Detailed Description

#### Date

2018/02/11

#### Author

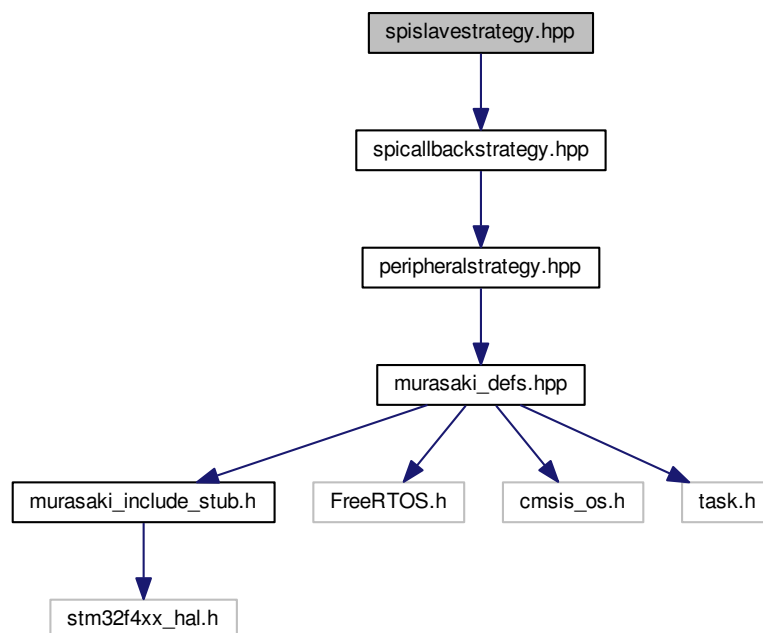
: Seiichi "Suikan" Horie

## 8.58 spislavestrategy.hpp File Reference

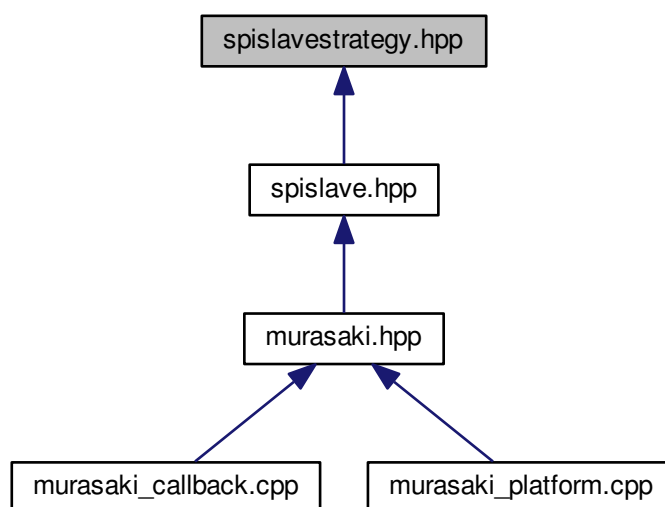
SPI slave root class.

```
#include <spicallbackstrategy.hpp>
```

Include dependency graph for spislavestrategy.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::SpiSlaveStrategy](#)  
*Root class of the SPI slave.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

### 8.58.1 Detailed Description

#### Date

2018/02/11

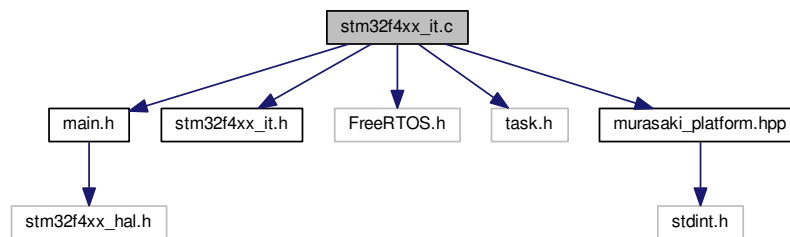
#### Author

: Seiichi "Suikan" Horie

## 8.59 stm32f4xx\_it.c File Reference

Interrupt Service Routines.

```
#include "main.h"
#include "stm32f4xx_it.h"
#include "FreeRTOS.h"
#include "task.h"
#include "murasaki_platform.hpp"
Include dependency graph for stm32f4xx_it.c:
```



## Functions

- void [NMI\\_Handler](#) (void)  
*This function handles Non maskable interrupt.*
- void [HardFault\\_Handler](#) (void)  
*This function handles Hard fault interrupt.*
- void [MemManage\\_Handler](#) (void)  
*This function handles Memory management fault.*
- void [BusFault\\_Handler](#) (void)  
*This function handles Pre-fetch fault, memory access fault.*
- void [UsageFault\\_Handler](#) (void)  
*This function handles Undefined instruction or illegal state.*
- void [DebugMon\\_Handler](#) (void)  
*This function handles Debug monitor.*
- void [DMA1\\_Stream3\\_IRQHandler](#) (void)  
*This function handles DMA1 stream3 global interrupt.*
- void [DMA1\\_Stream4\\_IRQHandler](#) (void)  
*This function handles DMA1 stream4 global interrupt.*
- void [DMA1\\_Stream5\\_IRQHandler](#) (void)  
*This function handles DMA1 stream5 global interrupt.*
- void [DMA1\\_Stream6\\_IRQHandler](#) (void)  
*This function handles DMA1 stream6 global interrupt.*
- void [TIM2\\_IRQHandler](#) (void)  
*This function handles TIM2 global interrupt.*
- void [I2C1\\_EV\\_IRQHandler](#) (void)  
*This function handles I2C1 event interrupt.*
- void [I2C1\\_ER\\_IRQHandler](#) (void)  
*This function handles I2C1 error interrupt.*



- void [I2C2\\_EV\\_IRQHandler](#) (void)  
*This function handles I2C2 event interrupt.*
- void [I2C2\\_ER\\_IRQHandler](#) (void)  
*This function handles I2C2 error interrupt.*
- void [SPI1\\_IRQHandler](#) (void)  
*This function handles SPI1 global interrupt.*
- void [SPI2\\_IRQHandler](#) (void)  
*This function handles SPI2 global interrupt.*
- void [USART1\\_IRQHandler](#) (void)  
*This function handles USART1 global interrupt.*
- void [USART2\\_IRQHandler](#) (void)  
*This function handles USART2 global interrupt.*
- void [TIM8\\_TRG\\_COM\\_TIM14\\_IRQHandler](#) (void)  
*This function handles TIM8 trigger and commutation interrupts and TIM14 global interrupt.*
- void [DMA2\\_Stream0\\_IRQHandler](#) (void)  
*This function handles DMA2 stream0 global interrupt.*
- void [DMA2\\_Stream2\\_IRQHandler](#) (void)  
*This function handles DMA2 stream2 global interrupt.*
- void [DMA2\\_Stream3\\_IRQHandler](#) (void)  
*This function handles DMA2 stream3 global interrupt.*
- void [DMA2\\_Stream7\\_IRQHandler](#) (void)  
*This function handles DMA2 stream7 global interrupt.*

#### Variables

- DMA\_HandleTypeDef [hdma\\_spi1\\_rx](#)

#### 8.59.1 Detailed Description

#### Attention

© Copyright (c) 2020 STMicroelectronics. All rights reserved.

This software component is licensed by ST under BSD 3-Clause license, the "License"; You may not use this file except in compliance with the License. You may obtain a copy of the License at: [opensource.org/licenses/BSD-3-Clause](https://opensource.org/licenses/BSD-3-Clause)

#### 8.59.2 Variable Documentation

##### 8.59.2.1 DMA\_HandleTypeDef hdma\_spi1\_rx

File Name : stm32f4xx\_hal\_msp.c Description : This file provides code for the MSP Initialization and de-Initialization codes.

#### Attention

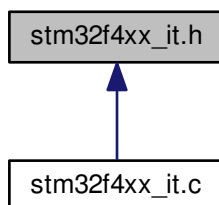
© Copyright (c) 2020 STMicroelectronics. All rights reserved.

This software component is licensed by ST under BSD 3-Clause license, the "License"; You may not use this file except in compliance with the License. You may obtain a copy of the License at: [opensource.org/licenses/BSD-3-Clause](https://opensource.org/licenses/BSD-3-Clause)

## 8.60 stm32f4xx\_it.h File Reference

This file contains the headers of the interrupt handlers.

This graph shows which files directly or indirectly include this file:



### Functions

- void [NMI\\_Handler](#) (void)  
*This function handles Non maskable interrupt.*
- void [HardFault\\_Handler](#) (void)  
*This function handles Hard fault interrupt.*
- void [MemManage\\_Handler](#) (void)  
*This function handles Memory management fault.*
- void [BusFault\\_Handler](#) (void)  
*This function handles Pre-fetch fault, memory access fault.*
- void [UsageFault\\_Handler](#) (void)  
*This function handles Undefined instruction or illegal state.*
- void [DebugMon\\_Handler](#) (void)  
*This function handles Debug monitor.*
- void [DMA1\\_Stream3\\_IRQHandler](#) (void)  
*This function handles DMA1 stream3 global interrupt.*
- void [DMA1\\_Stream4\\_IRQHandler](#) (void)  
*This function handles DMA1 stream4 global interrupt.*
- void [DMA1\\_Stream5\\_IRQHandler](#) (void)  
*This function handles DMA1 stream5 global interrupt.*
- void [DMA1\\_Stream6\\_IRQHandler](#) (void)  
*This function handles DMA1 stream6 global interrupt.*

- void [TIM2\\_IRQHandler](#) (void)  
*This function handles TIM2 global interrupt.*
- void [I2C1\\_EV\\_IRQHandler](#) (void)  
*This function handles I2C1 event interrupt.*
- void [I2C1\\_ER\\_IRQHandler](#) (void)  
*This function handles I2C1 error interrupt.*
- void [I2C2\\_EV\\_IRQHandler](#) (void)  
*This function handles I2C2 event interrupt.*
- void [I2C2\\_ER\\_IRQHandler](#) (void)  
*This function handles I2C2 error interrupt.*
- void [SPI1\\_IRQHandler](#) (void)  
*This function handles SPI1 global interrupt.*
- void [SPI2\\_IRQHandler](#) (void)  
*This function handles SPI2 global interrupt.*
- void [USART1\\_IRQHandler](#) (void)  
*This function handles USART1 global interrupt.*
- void [USART2\\_IRQHandler](#) (void)  
*This function handles USART2 global interrupt.*
- void [TIM8\\_TRG\\_COM\\_TIM14\\_IRQHandler](#) (void)  
*This function handles TIM8 trigger and commutation interrupts and TIM14 global interrupt.*
- void [DMA2\\_Stream0\\_IRQHandler](#) (void)  
*This function handles DMA2 stream0 global interrupt.*
- void [DMA2\\_Stream2\\_IRQHandler](#) (void)  
*This function handles DMA2 stream2 global interrupt.*
- void [DMA2\\_Stream3\\_IRQHandler](#) (void)  
*This function handles DMA2 stream3 global interrupt.*
- void [DMA2\\_Stream7\\_IRQHandler](#) (void)  
*This function handles DMA2 stream7 global interrupt.*

### 8.60.1 Detailed Description

#### Attention

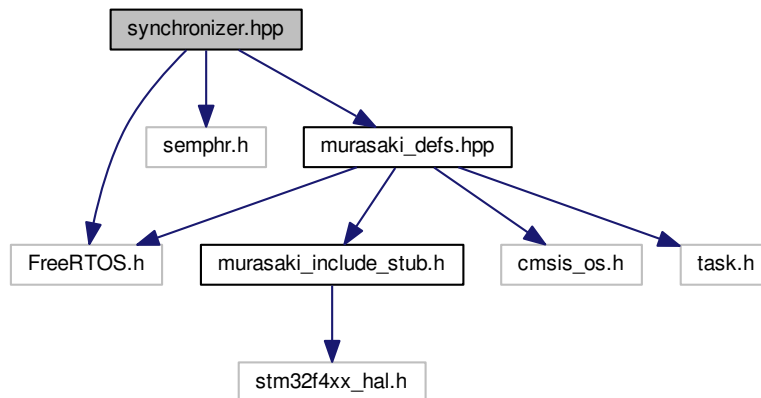
© Copyright (c) 2020 STMicroelectronics. All rights reserved.

This software component is licensed by ST under BSD 3-Clause license, the "License"; You may not use this file except in compliance with the License. You may obtain a copy of the License at: [opensource.org/licenses/BSD-3-Clause](https://opensource.org/licenses/BSD-3-Clause)

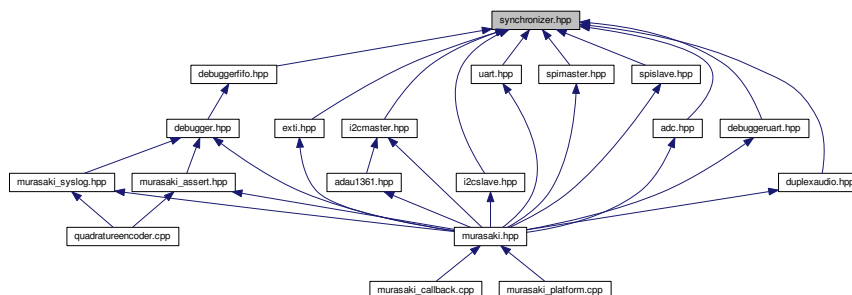
## 8.61 synchronizer.hpp File Reference

Synchronization between a Task and interrupt.

```
#include <FreeRTOS.h>
#include <semphr.h>
#include <murasaki_defs.hpp>
Include dependency graph for synchronizer.hpp:
```



This graph shows which files directly or indirectly include this file:



### Classes

- class [murasaki::Synchronizer](#)

*Synchronization class between a task and interrupt. This class provide the synchronization between a task and interrupt.*

### Namespaces

- [murasaki](#)

*Personal [Platform](#) parts collection.*

## 8.61.1 Detailed Description

## Date

2018/01/26

## Author

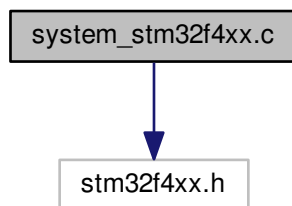
Seiichi "Suikan" Horie

## 8.62 system\_stm32f4xx.c File Reference

CMSIS Cortex-M4 Device Peripheral Access Layer System Source File.

```
#include "stm32f4xx.h"
```

Include dependency graph for system\_stm32f4xx.c:



## Macros

- `#define HSE_VALUE ((uint32_t)25000000)`
- `#define HSI_VALUE ((uint32_t)16000000)`
- `#define VECT_TAB_OFFSET 0x00`

## Functions

- void `SystemInit` (void)  
*Setup the microcontroller system Initialize the FPU setting, vector table location and External memory configuration.*
- void `SystemCoreClockUpdate` (void)  
*Update SystemCoreClock variable according to Clock Register Values. The SystemCoreClock variable contains the core clock (HCLK), it can be used by the user application to setup the SysTick timer or configure other parameters.*

### 8.62.1 Detailed Description

#### Author

MCD Application Team This file provides two functions and one global variable to be called from user application:

- **SystemInit()**: This function is called at startup just after reset and before branch to main program. This call is made inside the "startup\_stm32f4xx.s" file.
- **SystemCoreClock** variable: Contains the core clock (HCLK), it can be used by the user application to setup the SysTick timer or configure other parameters.
- **SystemCoreClockUpdate()**: Updates the variable SystemCoreClock and must be called whenever the core clock is changed during program execution.

#### Attention

© COPYRIGHT 2017 STMicroelectronics

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

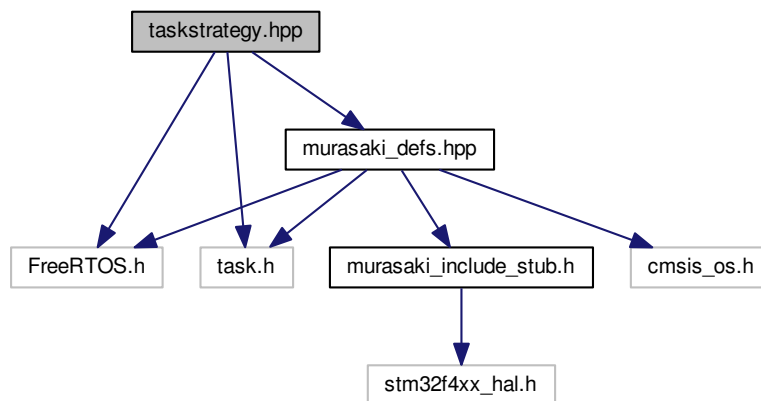
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of STMicroelectronics nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

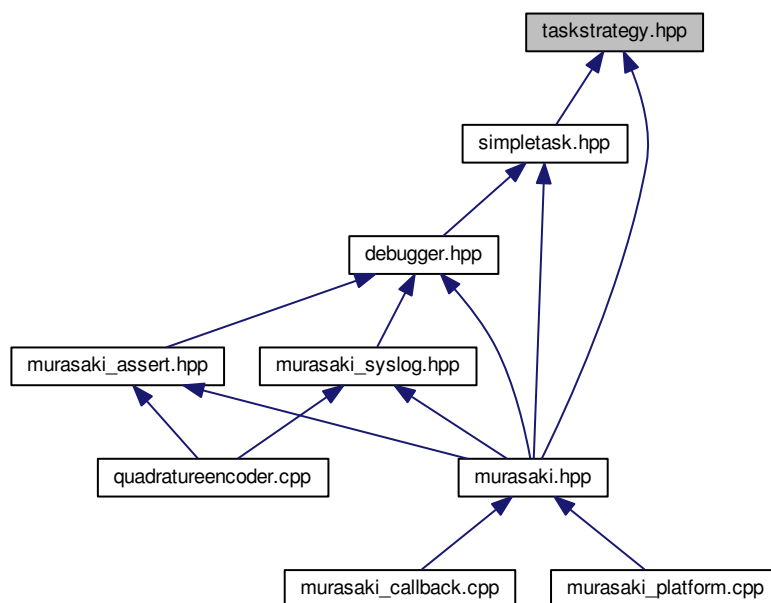
## 8.63 taskstrategy.hpp File Reference

Mother of All Tasks.

```
#include <FreeRTOS.h>
#include <task.h>
#include <murasaki_defs.hpp>
Include dependency graph for taskstrategy.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class [murasaki::TaskStrategy](#)  
*A mother of all tasks.*

## Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

## 8.63.1 Detailed Description

## Date

2018/02/20

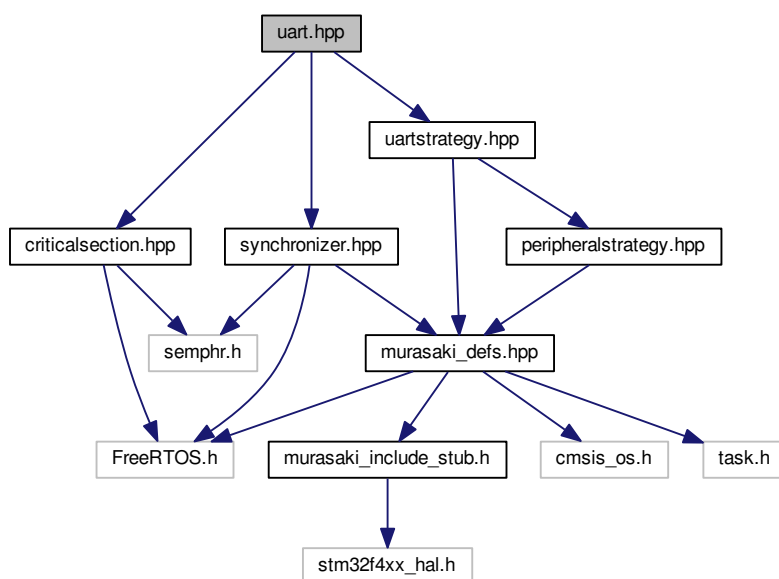
## Author

: Seiichi "Suikan" Horie

## 8.64 uart.hpp File Reference

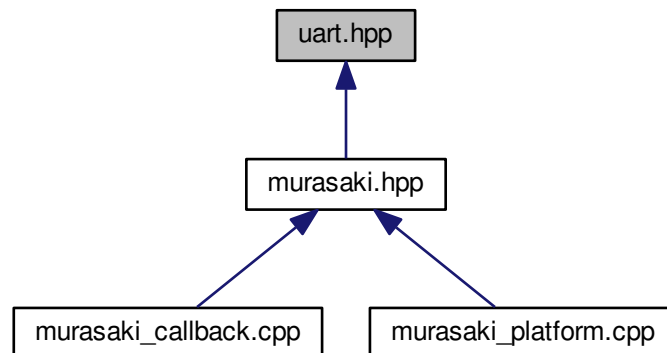
UART. Thread safe and synchronous IO.

```
#include <synchronizer.hpp>
#include <uartstrategy.hpp>
#include "criticalsection.hpp"
Include dependency graph for uart.hpp:
```





This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::Uart](#)

*Thread safe, synchronous and blocking IO. Concrete implementation of UART controller. Based on the STM32Cube HAL DMA Transfer.*

#### Namespaces

- [murasaki](#)

*Personal Platform parts collection.*

#### 8.64.1 Detailed Description

##### Date

2017/11/05

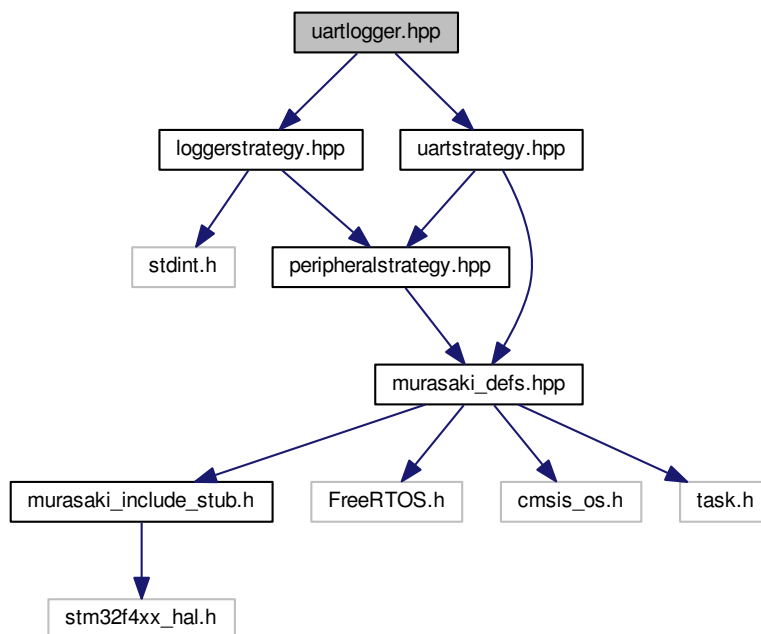
##### Author

Seiichi "Suikan" Horie

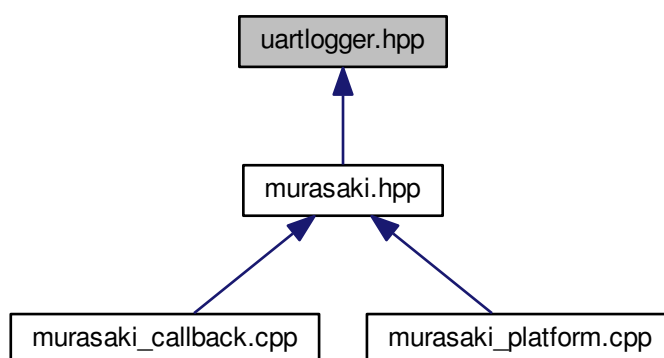
## 8.65 uartlogger.hpp File Reference

Logging to Uart.

```
#include <loggerstrategy.hpp>
#include <uartstrategy.hpp>
Include dependency graph for uartlogger.hpp:
```



This graph shows which files directly or indirectly include this file:



## Classes

- class `murasaki::UartLogger`  
*Logging through an UART port.*

## Namespaces

- [murasaki](#)

Personal [Platform](#) parts collection.

## 8.65.1 Detailed Description

## Date

2018/01/20

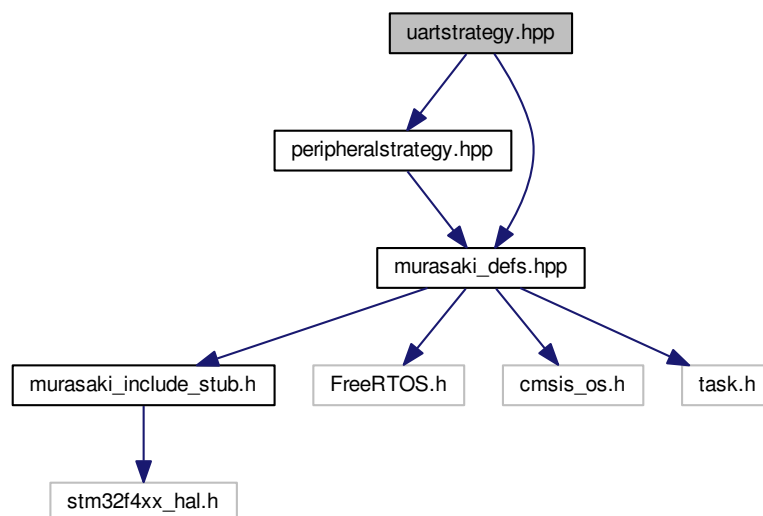
## Author

: Seiichi "Suikan" Horie

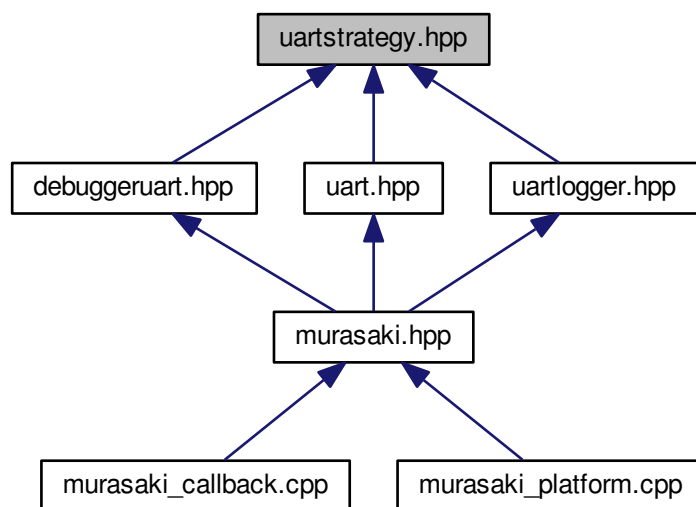
## 8.66 uartstrategy.hpp File Reference

Root class definition of the UART driver.

```
#include <peripheralstrategy.hpp>
#include "murasaki_defs.hpp"
Include dependency graph for uartstrategy.hpp:
```



This graph shows which files directly or indirectly include this file:



#### Classes

- class [murasaki::UartStrategy](#)  
*Definition of the root class of UART.*

#### Namespaces

- [murasaki](#)  
*Personal [Platform](#) parts collection.*

#### 8.66.1 Detailed Description

##### Date

2017/11/04

##### Author

: Seiichi "Suikan" Horie

## Index

- ~LoggerStrategy
  - [murasaki::LoggerStrategy](#), [152](#)
- Abstract Classes, [47](#)
- Adau1361
  - [murasaki::Adau1361](#), [70](#)
- [adau1361.hpp](#), [209](#)
- Adc
  - [murasaki::Adc](#), [76](#)
- [adc.hpp](#), [210](#)
- AdcStatus
  - Definitions and Configuration, [35](#)
- [adcstrategy.hpp](#), [212](#)
- AddExtiObject
  - [murasaki::ExtiCallbackRepositorySingleton](#), [121](#)
- AddPeripheralObject
  - [murasaki::CallbackRepositorySingleton](#), [98](#)
- AddSyslogFacilityToMask
  - [murasaki](#), [66](#)
- [allocators.cpp](#), [213](#)
- AllowedSyslogOut
  - [murasaki](#), [67](#)
- Application Specific Platform, [40](#)
  - CustomAssertFailed, [41](#)
  - CustomDefaultHandler, [42](#)
  - debugger, [46](#)
  - ExecPlatform, [42](#)
  - HAL\_GPIO\_EXTI\_Callback, [42](#)
  - HAL\_I2C\_ErrorCallback, [43](#)
  - HAL\_I2C\_MasterTxCpltCallback, [43](#)
  - HAL\_I2C\_SlaveTxCpltCallback, [43](#)
  - HAL\_I2S\_ErrorCallback, [44](#)
  - HAL\_I2S\_RxCpltCallback, [44](#)
  - HAL\_I2S\_RxHalfCpltCallback, [44](#)
  - HAL\_SAI\_ErrorCallback, [44](#)
  - HAL\_SAI\_RxCpltCallback, [44](#)
  - HAL\_SAI\_RxHalfCpltCallback, [45](#)
  - HAL\_SPI\_ErrorCallback, [45](#)
  - HAL\_SPI\_TxRxCpltCallback, [45](#)
  - HAL\_UART\_ErrorCallback, [45](#)
  - HAL\_UART\_RxCpltCallback, [46](#)
  - HAL\_UART\_TxCpltCallback, [46](#)
  - InitPlatform, [46](#)
- assert\_failed
  - [main.c](#), [252](#)
- AssertCs
  - [murasaki::SpiSlaveAdapter](#), [187](#)
  - [murasaki::SpiSlaveAdapterStrategy](#), [189](#)
- AudioCodecStrategy
  - [murasaki::AudioCodecStrategy](#), [82](#)
- [audiocodecstrategy.hpp](#), [214](#)
- [audioportadapterstrategy.hpp](#), [216](#)
- AutoRePrint
  - [murasaki::Debugger](#), [101](#)
- BitIn
  - [murasaki::BitIn](#), [90](#)
- BitOut
  - [murasaki::BitOut](#), [94](#)
- [bitin.hpp](#), [217](#)
- [bitinstrategy.hpp](#), [218](#)
- [bitout.hpp](#), [220](#)
- [bitoutstrategy.hpp](#), [222](#)
- CMSIS, [54](#)
- [callbackrepositorysingleton.hpp](#), [224](#)
- Clear
  - [murasaki::BitOutStrategy](#), [97](#)
- CodecChannel
  - Definitions and Configuration, [35](#)
- ConfigurePll
  - [murasaki::Adau1361](#), [71](#)
- ConversionCompleteCallback
  - [murasaki::Adc](#), [76](#)
  - [murasaki::AdcStrategy](#), [79](#)
- Convert
  - [murasaki::Adc](#), [76](#)
  - [murasaki::AdcStrategy](#), [79](#)
- [criticalsection.hpp](#), [225](#)
- CustomAssertFailed
  - Application Specific Platform, [41](#)
- CustomDefaultHandler
  - Application Specific Platform, [42](#)
- DeassertCs
  - [murasaki::SpiSlaveAdapter](#), [187](#)
  - [murasaki::SpiSlaveAdapterStrategy](#), [189](#)
- Debugger
  - [murasaki::Debugger](#), [101](#)
- debugger
  - Application Specific Platform, [46](#)
- [debugger.hpp](#), [226](#)
- DebuggerFifo
  - [murasaki::DebuggerFifo](#), [104](#)
- DebuggerUart
  - [murasaki::DebuggerUart](#), [106](#)
- [debuggerfifo.hpp](#), [228](#)
- [debuggeruart.hpp](#), [230](#)
- Definitions and Configuration, [33](#)
  - AdcStatus, [35](#)
  - CodecChannel, [35](#)
  - I2cStatus, [35](#)
  - InterruptStatus, [35](#)
  - kasOK, [35](#)
  - kccAuxInput, [35](#)
  - kccHeadphoneOutput, [35](#)
  - kccLineInput, [35](#)
  - kccLineOutput, [35](#)
  - kccMicInput, [35](#)
  - kfaAdc, [37](#)
  - kfaAll, [37](#)
  - kfaAudio, [37](#)

- kfaAudioCodec, [37](#)
- kfaEncoder, [37](#)
- kfaExti, [37](#)
- kfaI2cMaster, [37](#)
- kfaI2cSlave, [37](#)
- kfaI2s, [37](#)
- kfaKernel, [37](#)
- kfaLog, [37](#)
- kfaNone, [37](#)
- kfaSai, [37](#)
- kfaSerial, [37](#)
- kfaSpiMaster, [37](#)
- kfaSpiSlave, [37](#)
- kfaUser0, [37](#)
- kfaUser1, [37](#)
- kfaUser2, [37](#)
- kfaUser3, [37](#)
- kfaUser4, [37](#)
- kfaUser5, [37](#)
- kfaUser6, [37](#)
- kfaUser7, [37](#)
- ki2csArbitrationLost, [35](#)
- ki2csBussError, [35](#)
- ki2csDMA, [35](#)
- ki2csNak, [35](#)
- ki2csOK, [35](#)
- ki2csOverrun, [35](#)
- ki2csTimeOut, [35](#)
- ki2csUnknown, [35](#)
- kisOK, [36](#)
- kisTimeOut, [36](#)
- kseAlert, [37](#)
- kseCritical, [37](#)
- kseDebug, [37](#)
- kseEmergency, [37](#)
- kseError, [37](#)
- kseInfomational, [37](#)
- kseNotice, [37](#)
- kseWarning, [37](#)
- ksphLatchThenShift, [36](#)
- ksphShiftThenLatch, [36](#)
- kspisAbort, [36](#)
- kspisDMA, [36](#)
- kspisErrorFlag, [36](#)
- kspisFrameError, [36](#)
- kspisModeCRC, [36](#)
- kspisModeFault, [36](#)
- kspisOK, [36](#)
- kspisOverflow, [36](#)
- kspisTimeOut, [36](#)
- kspisUnknown, [36](#)
- kspoFallThenRise, [36](#)
- kspoRiseThenFall, [36](#)
- ktpAboveNormal, [38](#)
- ktpBelowNormal, [38](#)
- ktpHigh, [38](#)
- ktpIdle, [38](#)
- ktpLow, [38](#)
- ktpNormal, [38](#)
- ktpRealtime, [38](#)
- kuhfcCts, [38](#)
- kuhfcCtsRts, [38](#)
- kuhfcNone, [38](#)
- kuhfcRts, [38](#)
- kursDMA, [38](#)
- kursFrame, [38](#)
- kursNoise, [38](#)
- kursOK, [38](#)
- kursOverrun, [38](#)
- kursParity, [38](#)
- kursTimeOut, [38](#)
- kutIdleTimeout, [39](#)
- kutNoldleTimeout, [39](#)
- kwmsIndefinitely, [39](#)
- kwmsPolling, [39](#)
- MURASAKI\_CONFIG\_NOCYCCNT, [33](#)
- MURASAKI\_CONFIG\_NODEBUG, [33](#)
- NUM\_OF\_CALLBACK\_OBJECTS, [33](#)
- NUM\_OF\_EXTI\_OBJECTS, [34](#)
- PLATFORM\_CONFIG\_DEBUG\_BUFFER\_SIZE, [34](#)
- PLATFORM\_CONFIG\_DEBUG\_LINE\_SIZE, [34](#)
- PLATFORM\_CONFIG\_DEBUG\_SERIAL\_TIME↔OUT, [34](#)
- PLATFORM\_CONFIG\_DEBUG\_TASK\_PRIORITY, [34](#)
- PLATFORM\_CONFIG\_DEBUG\_TASK\_STACK↔\_SIZE, [34](#)
- SpiClockPhase, [36](#)
- SpiClockPolarity, [36](#)
- SpiStatus, [36](#)
- SyslogFacility, [36](#)
- SyslogSeverity, [37](#)
- TaskPriority, [37](#)
- UartHardwareFlowControl, [38](#)
- UartStatus, [38](#)
- UartTimeout, [38](#)
- WaitMilliSeconds, [39](#)
- DetectPhase
  - murasaki::AudioPortAdapterStrategy, [85](#)
- DmaCallback
  - murasaki::DuplexAudio, [112](#)
- DoPostMortem
  - murasaki::LoggerStrategy, [152](#)
  - murasaki::UartLogger, [204](#)
- DuplexAudio
  - murasaki::DuplexAudio, [111](#)
- duplexaudio.hpp, [231](#)
- Enter
  - murasaki::CriticalSection, [99](#)
- Error\_Handler
  - main.c, [252](#)
  - main.h, [255](#)
- ExecPlatform
  - Application Specific Platform, [42](#)
- Exti

- [murasaki::Exti](#), [118](#)
- [exti.hpp](#), [232](#)
- [exticallbackrepositorysingleton.hpp](#), [234](#)
- [facility\\_mask\\_](#)
  - [murasaki::Debugger](#), [102](#)
- [FifoStrategy](#)
  - [murasaki::FifoStrategy](#), [122](#)
- [fifostrategy.hpp](#), [235](#)
- [Get](#)
  - [murasaki::BitIn](#), [90](#)
  - [murasaki::BitInStrategy](#), [92](#)
  - [murasaki::BitOut](#), [95](#)
  - [murasaki::BitOutStrategy](#), [97](#)
  - [murasaki::DebuggerFifo](#), [104](#)
  - [murasaki::FifoStrategy](#), [123](#)
  - [murasaki::QuadratureEncoder](#), [159](#)
  - [murasaki::QuadratureEncoderStrategy](#), [161](#)
- [getCharacter](#)
  - [murasaki::LoggerStrategy](#), [153](#)
  - [murasaki::UartLogger](#), [204](#)
- [GetCpha](#)
  - [murasaki::SpiSlaveAdapterStrategy](#), [189](#)
- [GetCpol](#)
  - [murasaki::SpiSlaveAdapterStrategy](#), [189](#)
- [GetCycleCounter](#)
  - Utility functions, [52](#)
- [GetExtiObject](#)
  - [murasaki::ExtiCallbackRepositorySingleton](#), [121](#)
- [GetInstance](#)
  - [murasaki::CallbackRepositorySingleton](#), [98](#)
  - [murasaki::ExtiCallbackRepositorySingleton](#), [121](#)
- [GetName](#)
  - [murasaki::TaskStrategy](#), [194](#)
- [GetNumberOfChannelsRx](#)
  - [murasaki::AudioPortAdapterStrategy](#), [85](#)
  - [murasaki::I2sPortAdapter](#), [146](#)
  - [murasaki::SaiPortAdapter](#), [166](#)
- [GetNumberOfChannelsTx](#)
  - [murasaki::AudioPortAdapterStrategy](#), [85](#)
  - [murasaki::I2sPortAdapter](#), [146](#)
  - [murasaki::SaiPortAdapter](#), [166](#)
- [GetNumberOfDMAPhase](#)
  - [murasaki::AudioPortAdapterStrategy](#), [85](#)
  - [murasaki::I2sPortAdapter](#), [147](#)
  - [murasaki::SaiPortAdapter](#), [166](#)
- [GetPeripheralHandle](#)
  - [murasaki::Adc](#), [77](#)
  - [murasaki::AudioPortAdapterStrategy](#), [85](#)
  - [murasaki::BitIn](#), [90](#)
  - [murasaki::BitOut](#), [95](#)
  - [murasaki::DuplexAudio](#), [113](#)
  - [murasaki::I2sPortAdapter](#), [147](#)
  - [murasaki::PeripheralStrategy](#), [155](#)
  - [murasaki::SaiPortAdapter](#), [166](#)
- [GetPeripheralObject](#)
  - [murasaki::CallbackRepositorySingleton](#), [98](#)
- [GetSampleShiftSizeRx](#)
  - [murasaki::AudioPortAdapterStrategy](#), [85](#)
  - [murasaki::I2sPortAdapter](#), [147](#)
  - [murasaki::SaiPortAdapter](#), [166](#)
- [GetSampleShiftSizeTx](#)
  - [murasaki::AudioPortAdapterStrategy](#), [86](#)
  - [murasaki::I2sPortAdapter](#), [147](#)
  - [murasaki::SaiPortAdapter](#), [167](#)
- [GetSampleWordSizeRx](#)
  - [murasaki::AudioPortAdapterStrategy](#), [86](#)
  - [murasaki::I2sPortAdapter](#), [148](#)
  - [murasaki::SaiPortAdapter](#), [167](#)
- [GetSampleWordSizeTx](#)
  - [murasaki::AudioPortAdapterStrategy](#), [86](#)
  - [murasaki::I2sPortAdapter](#), [148](#)
  - [murasaki::SaiPortAdapter](#), [167](#)
- [getStackDepth](#)
  - [murasaki::TaskStrategy](#), [194](#)
- [getStackMinHeadroom](#)
  - [murasaki::TaskStrategy](#), [194](#)
- [GetchFromTask](#)
  - [murasaki::Debugger](#), [101](#)
- [HAL\\_ADC\\_ConvCpltCallback](#)
  - [murasaki\\_callback.cpp](#), [260](#)
- [HAL\\_ADC\\_ErrorCallback](#)
  - [murasaki\\_callback.cpp](#), [261](#)
- [HAL\\_GPIO\\_EXTI\\_Callback](#)
  - Application Specific Platform, [42](#)
- [HAL\\_I2C\\_ErrorCallback](#)
  - Application Specific Platform, [43](#)
- [HAL\\_I2C\\_MasterRxCpltCallback](#)
  - [murasaki\\_callback.cpp](#), [261](#)
- [HAL\\_I2C\\_MasterTxCpltCallback](#)
  - Application Specific Platform, [43](#)
- [HAL\\_I2C\\_SlaveRxCpltCallback](#)
  - [murasaki\\_callback.cpp](#), [261](#)
- [HAL\\_I2C\\_SlaveTxCpltCallback](#)
  - Application Specific Platform, [43](#)
- [HAL\\_I2S\\_ErrorCallback](#)
  - Application Specific Platform, [44](#)
- [HAL\\_I2S\\_RxCpltCallback](#)
  - Application Specific Platform, [44](#)
- [HAL\\_I2S\\_RxHalfCpltCallback](#)
  - Application Specific Platform, [44](#)
- [HAL\\_SAI\\_ErrorCallback](#)
  - Application Specific Platform, [44](#)
- [HAL\\_SAI\\_RxCpltCallback](#)
  - Application Specific Platform, [44](#)
- [HAL\\_SAI\\_RxHalfCpltCallback](#)
  - Application Specific Platform, [45](#)
- [HAL\\_SPI\\_ErrorCallback](#)
  - Application Specific Platform, [45](#)
- [HAL\\_SPI\\_TxRxCpltCallback](#)
  - Application Specific Platform, [45](#)
- [HAL\\_TIM\\_PeriodElapsedCallback](#)
  - [main.c](#), [252](#)
- [HAL\\_UART\\_ErrorCallback](#)
  - Application Specific Platform, [45](#)
- [HAL\\_UART\\_RxCpltCallback](#)

- Application Specific Platform, [46](#)
- HAL\_UART\_TxCpltCallback
  - Application Specific Platform, [46](#)
- HSE\_VALUE
  - STM32F4xx\_System\_Private\_Includes, [56](#)
- HSI\_VALUE
  - STM32F4xx\_System\_Private\_Includes, [56](#)
- HandleError
  - murasaki::Adc, [77](#)
  - murasaki::AdcStrategy, [79](#)
  - murasaki::AudioPortAdapterStrategy, [86](#)
  - murasaki::DebuggerUart, [107](#)
  - murasaki::DuplexAudio, [113](#)
  - murasaki::I2cCallbackStrategy, [125](#)
  - murasaki::I2cMaster, [129](#)
  - murasaki::I2cSlave, [137](#)
  - murasaki::I2sPortAdapter, [148](#)
  - murasaki::SaiPortAdapter, [167](#)
  - murasaki::SpiCallbackStrategy, [172](#)
  - murasaki::SpiMaster, [176](#)
  - murasaki::SpiSlave, [183](#)
  - murasaki::Uart, [198](#)
  - murasaki::UartStrategy, [206](#)
- hdma\_spi1\_rx
  - main.c, [253](#)
  - stm32f4xx\_it.c, [291](#)
- Helper classes, [50](#)
  - operator delete, [50](#)
  - operator delete[], [51](#)
  - operator new, [51](#)
  - operator new[], [51](#)
- I2cMaster
  - murasaki::I2cMaster, [129](#)
- I2cSearch
  - Utility functions, [52](#)
- I2cStatus
  - Definitions and Configuration, [35](#)
- i2ccallbackstrategy.hpp, [237](#)
- i2cmaster.hpp, [239](#)
- i2cmasterstrategy.hpp, [241](#)
- i2cslave.hpp, [243](#)
- i2cslavestrategy.hpp, [244](#)
- I2sPortAdapter
  - murasaki::I2sPortAdapter, [146](#)
- i2sportadapter.hpp, [246](#)
- InitCycleCounter
  - Utility functions, [52](#)
- InitPlatform
  - Application Specific Platform, [46](#)
- InterruptStatus
  - Definitions and Configuration, [35](#)
- interruptstrategy.hpp, [247](#)
- IsInt16SwapRequired
  - murasaki::AudioPortAdapterStrategy, [87](#)
  - murasaki::I2sPortAdapter, [149](#)
  - murasaki::SaiPortAdapter, [168](#)
- isReady
  - murasaki::Exti, [119](#)
- kasOK
  - Definitions and Configuration, [35](#)
- kccAuxInput
  - Definitions and Configuration, [35](#)
- kccHeadphoneOutput
  - Definitions and Configuration, [35](#)
- kccLineInput
  - Definitions and Configuration, [35](#)
- kccLineOutput
  - Definitions and Configuration, [35](#)
- kccMicInput
  - Definitions and Configuration, [35](#)
- kfaAdc
  - Definitions and Configuration, [37](#)
- kfaAll
  - Definitions and Configuration, [37](#)
- kfaAudio
  - Definitions and Configuration, [37](#)
- kfaAudioCodec
  - Definitions and Configuration, [37](#)
- kfaEncoder
  - Definitions and Configuration, [37](#)
- kfaExti
  - Definitions and Configuration, [37](#)
- kfal2cMaster
  - Definitions and Configuration, [37](#)
- kfal2cSlave
  - Definitions and Configuration, [37](#)
- kfal2s
  - Definitions and Configuration, [37](#)
- kfaKernel
  - Definitions and Configuration, [37](#)
- kfaLog
  - Definitions and Configuration, [37](#)
- kfaNone
  - Definitions and Configuration, [37](#)
- kfaSai
  - Definitions and Configuration, [37](#)
- kfaSerial
  - Definitions and Configuration, [37](#)
- kfaSpiMaster
  - Definitions and Configuration, [37](#)
- kfaSpiSlave
  - Definitions and Configuration, [37](#)
- kfaUser0
  - Definitions and Configuration, [37](#)
- kfaUser1
  - Definitions and Configuration, [37](#)
- kfaUser2
  - Definitions and Configuration, [37](#)
- kfaUser3
  - Definitions and Configuration, [37](#)
- kfaUser4
  - Definitions and Configuration, [37](#)
- kfaUser5
  - Definitions and Configuration, [37](#)
- kfaUser6
  - Definitions and Configuration, [37](#)



- kfaUser7
  - Definitions and Configuration, [37](#)
- ki2csArbitrationLost
  - Definitions and Configuration, [35](#)
- ki2csBussError
  - Definitions and Configuration, [35](#)
- ki2csDMA
  - Definitions and Configuration, [35](#)
- ki2csNak
  - Definitions and Configuration, [35](#)
- ki2csOK
  - Definitions and Configuration, [35](#)
- ki2csOverrun
  - Definitions and Configuration, [35](#)
- ki2csTimeOut
  - Definitions and Configuration, [35](#)
- ki2csUnknown
  - Definitions and Configuration, [35](#)
- kisOK
  - Definitions and Configuration, [36](#)
- kisTimeOut
  - Definitions and Configuration, [36](#)
- kseAlert
  - Definitions and Configuration, [37](#)
- kseCritical
  - Definitions and Configuration, [37](#)
- kseDebug
  - Definitions and Configuration, [37](#)
- kseEmergency
  - Definitions and Configuration, [37](#)
- kseError
  - Definitions and Configuration, [37](#)
- kseInfomational
  - Definitions and Configuration, [37](#)
- kseNotice
  - Definitions and Configuration, [37](#)
- kseWarning
  - Definitions and Configuration, [37](#)
- ksphLatchThenShift
  - Definitions and Configuration, [36](#)
- ksphShiftThenLatch
  - Definitions and Configuration, [36](#)
- kspisAbort
  - Definitions and Configuration, [36](#)
- kspisDMA
  - Definitions and Configuration, [36](#)
- kspisErrorFlag
  - Definitions and Configuration, [36](#)
- kspisFrameError
  - Definitions and Configuration, [36](#)
- kspisModeCRC
  - Definitions and Configuration, [36](#)
- kspisModeFault
  - Definitions and Configuration, [36](#)
- kspisOK
  - Definitions and Configuration, [36](#)
- kspisOverflow
  - Definitions and Configuration, [36](#)
- kspisTimeOut
  - Definitions and Configuration, [36](#)
- kspisUnknown
  - Definitions and Configuration, [36](#)
- kspoFallThenRise
  - Definitions and Configuration, [36](#)
- kspoRiseThenFall
  - Definitions and Configuration, [36](#)
- ktpAboveNormal
  - Definitions and Configuration, [38](#)
- ktpBelowNormal
  - Definitions and Configuration, [38](#)
- ktpHigh
  - Definitions and Configuration, [38](#)
- ktpIdle
  - Definitions and Configuration, [38](#)
- ktpLow
  - Definitions and Configuration, [38](#)
- ktpNormal
  - Definitions and Configuration, [38](#)
- ktpRealtime
  - Definitions and Configuration, [38](#)
- kuhfcCts
  - Definitions and Configuration, [38](#)
- kuhfcCtsRts
  - Definitions and Configuration, [38](#)
- kuhfcNone
  - Definitions and Configuration, [38](#)
- kuhfcRts
  - Definitions and Configuration, [38](#)
- kursDMA
  - Definitions and Configuration, [38](#)
- kursFrame
  - Definitions and Configuration, [38](#)
- kursNoise
  - Definitions and Configuration, [38](#)
- kursOK
  - Definitions and Configuration, [38](#)
- kursOverrun
  - Definitions and Configuration, [38](#)
- kursParity
  - Definitions and Configuration, [38](#)
- kursTimeOut
  - Definitions and Configuration, [38](#)
- kutIdleTimeout
  - Definitions and Configuration, [39](#)
- kutNoldleTimeout
  - Definitions and Configuration, [39](#)
- kwmsIndefinitely
  - Definitions and Configuration, [39](#)
- kwmsPolling
  - Definitions and Configuration, [39](#)
- Launch
  - murasaki::TaskStrategy, [194](#)
- Leave
  - murasaki::CriticalSection, [99](#)
- line\_
  - murasaki::Debugger, [102](#)

- loggerstrategy.hpp, 249
- MURASAKI\_ASSERT
  - Murasaki Class Collection, 29
- MURASAKI\_CONFIG\_NOCYCNT
  - Definitions and Configuration, 33
- MURASAKI\_CONFIG\_NODEBUG
  - Definitions and Configuration, 33
- MURASAKI\_CONFIG\_NOSYSLOG
  - platform\_config.hpp, 271
- MURASAKI\_PRINT\_ERROR
  - Murasaki Class Collection, 29
- MURASAKI\_SYSLOG
  - Murasaki Class Collection, 30
- main
  - main.c, 252
- main.c, 251
  - assert\_failed, 252
  - Error\_Handler, 252
  - HAL\_TIM\_PeriodElapsedCallback, 252
  - hdma\_spi1\_rx, 253
  - main, 252
  - StartDefaultTask, 253
  - SystemClock\_Config, 253
- main.h, 254
  - Error\_Handler, 255
- MasterTaskBodyFunction
  - murasaki\_platform.hpp, 267
- Match
  - murasaki::AudioPortAdapterStrategy, 87
  - murasaki::DuplexAudio, 113
  - murasaki::Exti, 119
  - murasaki::I2sPortAdapter, 149
  - murasaki::InterruptStrategy, 150
  - murasaki::PeripheralStrategy, 155
  - murasaki::SaiPortAdapter, 168
- murasaki, 64
  - AddSyslogFacilityToMask, 66
  - AllowedSyslogOut, 67
  - platform, 68
  - RemoveSyslogFacilityFromMask, 67
  - SetSyslogFacilityMask, 67
  - SetSyslogSererityThreshold, 67
- Murasaki API reference, 26
- Murasaki Class Collection, 28
  - MURASAKI\_ASSERT, 29
  - MURASAKI\_PRINT\_ERROR, 29
  - MURASAKI\_SYSLOG, 30
- murasaki.hpp, 255
- murasaki::Adau1361, 68
  - Adau1361, 70
  - ConfigurePll, 71
  - Mute, 71
  - SendCommand, 71
  - SendCommandTable, 71
  - SetAuxInputGain, 72
  - SetGain, 72
  - SetHpOutputGain, 72
  - SetLineInputGain, 72
  - SetLineOutputGain, 73
  - Start, 73
  - WaitPllLock, 73
- murasaki::Adc, 74
  - Adc, 76
  - ConversionCompleteCallback, 76
  - Convert, 76
  - GetPeripheralHandle, 77
  - HandleError, 77
  - SetSampleClock, 77
- murasaki::AdcStrategy, 78
  - ConversionCompleteCallback, 79
  - Convert, 79
  - HandleError, 79
  - SetSampleClock, 81
- murasaki::AudioCodecStrategy, 81
  - AudioCodecStrategy, 82
  - Mute, 82
  - SendCommand, 82
  - SetGain, 83
  - Start, 83
- murasaki::AudioPortAdapterStrategy, 83
  - DetectPhase, 85
  - GetNumberOfChannelsRx, 85
  - GetNumberOfChannelsTx, 85
  - GetNumberOfDMAPhase, 85
  - GetPeripheralHandle, 85
  - GetSampleShiftSizeRx, 85
  - GetSampleShiftSizeTx, 86
  - GetSampleWordSizeRx, 86
  - GetSampleWordSizeTx, 86
  - HandleError, 86
  - IsInt16SwapRequired, 87
  - Match, 87
  - StartTransferRx, 87
  - StartTransferTx, 87
- murasaki::BitIn, 88
  - BitIn, 90
  - Get, 90
  - GetPeripheralHandle, 90
- murasaki::BitInStrategy, 91
  - Get, 92
- murasaki::BitOut, 92
  - BitOut, 94
  - Get, 95
  - GetPeripheralHandle, 95
  - Set, 95
- murasaki::BitOutStrategy, 95
  - Clear, 97
  - Get, 97
  - Set, 97
- murasaki::CallbackRepositorySingleton, 97
  - AddPeripheralObject, 98
  - GetInstance, 98
  - GetPeripheralObject, 98
- murasaki::CriticalSection, 99
  - Enter, 99
  - Leave, 99

- murasaki::Debugger, 100
  - AutoRePrint, 101
  - Debugger, 101
  - facility\_mask\_, 102
  - GetchFromTask, 101
  - line\_, 102
  - Printf, 102
  - RePrint, 102
  - severity\_, 102
- murasaki::DebuggerFifo, 103
  - DebuggerFifo, 104
  - Get, 104
  - SetPostMortem, 105
- murasaki::DebuggerUart, 105
  - DebuggerUart, 106
  - HandleError, 107
  - Receive, 107
  - ReceiveCompleteCallback, 107
  - SetHardwareFlowControl, 108
  - SetSpeed, 108
  - Transmit, 108
  - TransmitCompleteCallback, 109
- murasaki::DuplexAudio, 110
  - DmaCallback, 112
  - DuplexAudio, 111
  - GetPeripheralHandle, 113
  - HandleError, 113
  - Match, 113
  - TransmitAndReceive, 114
- murasaki::Exti, 115
  - Exti, 118
  - isReady, 119
  - Match, 119
  - Release, 119
  - Wait, 120
- murasaki::ExtiCallbackRepositorySingleton, 120
  - AddExtiObject, 121
  - GetExtiObject, 121
  - GetInstance, 121
- murasaki::FifoStrategy, 122
  - FifoStrategy, 122
  - Get, 123
  - Put, 123
- murasaki::GPIO\_type, 123
- murasaki::I2cCallbackStrategy, 124
  - HandleError, 125
  - ReceiveCompleteCallback, 125
  - TransmitCompleteCallback, 125
- murasaki::I2cMaster, 126
  - HandleError, 129
  - I2cMaster, 129
  - Receive, 130
  - ReceiveCompleteCallback, 130
  - Transmit, 131
  - TransmitCompleteCallback, 131
  - TransmitThenReceive, 132
- murasaki::I2cMasterStrategy, 133
  - Receive, 134
  - Transmit, 134
  - TransmitThenReceive, 135
- murasaki::I2cSlave, 136
  - HandleError, 137
  - Receive, 138
  - ReceiveCompleteCallback, 138
  - Transmit, 139
  - TransmitCompleteCallback, 139
- murasaki::I2cSlaveStrategy, 140
  - Receive, 141
  - Transmit, 142
- murasaki::I2sPortAdapter, 142
  - GetNumberOfChannelsRx, 146
  - GetNumberOfChannelsTx, 146
  - GetNumberOfDMAPhase, 147
  - GetPeripheralHandle, 147
  - GetSampleShiftSizeRx, 147
  - GetSampleShiftSizeTx, 147
  - GetSampleWordSizeRx, 148
  - GetSampleWordSizeTx, 148
  - HandleError, 148
  - I2sPortAdapter, 146
  - IsInt16SwapRequired, 149
  - Match, 149
  - StartTransferRx, 149
  - StartTransferTx, 149
- murasaki::InterruptStrategy, 150
  - Match, 150
  - Release, 151
  - Wait, 151
- murasaki::LoggerStrategy, 151
  - ~LoggerStrategy, 152
  - DoPostMortem, 152
  - getCharacter, 153
  - putMessage, 153
- murasaki::LoggingHelpers, 153
- murasaki::PeripheralStrategy, 154
  - GetPeripheralHandle, 155
  - Match, 155
- murasaki::Platform, 156
- murasaki::QuadratureEncoder, 157
  - Get, 159
  - QuadratureEncoder, 158
  - Set, 159
- murasaki::QuadratureEncoderStrategy, 159
  - Get, 161
  - Set, 161
- murasaki::SaiPortAdapter, 161
  - GetNumberOfChannelsRx, 166
  - GetNumberOfChannelsTx, 166
  - GetNumberOfDMAPhase, 166
  - GetPeripheralHandle, 166
  - GetSampleShiftSizeRx, 166
  - GetSampleShiftSizeTx, 167
  - GetSampleWordSizeRx, 167
  - GetSampleWordSizeTx, 167
  - HandleError, 167
  - IsInt16SwapRequired, 168

- Match, 168
- SaiPortAdapter, 165
- StartTransferRx, 168
- StartTransferTx, 168
- murasaki::SimpleTask, 169
  - SimpleTask, 170
  - TaskBody, 171
- murasaki::SpiCallbackStrategy, 171
  - HandleError, 172
  - TransmitAndReceiveCompleteCallback, 172
- murasaki::SpiMaster, 173
  - HandleError, 176
  - SpiMaster, 176
  - TransmitAndReceive, 177
  - TransmitAndReceiveCompleteCallback, 177
- murasaki::SpiMasterStrategy, 178
  - TransmitAndReceive, 179
- murasaki::SpiSlave, 180
  - HandleError, 183
  - SpiSlave, 183
  - TransmitAndReceive, 184
  - TransmitAndReceiveCompleteCallback, 184
- murasaki::SpiSlaveAdapter, 185
  - AssertCs, 187
  - DeassertCs, 187
  - SpiSlaveAdapter, 186, 187
- murasaki::SpiSlaveAdapterStrategy, 187
  - AssertCs, 189
  - DeassertCs, 189
  - GetCpha, 189
  - GetCpol, 189
  - SpiSlaveAdapterStrategy, 188, 189
- murasaki::SpiSlaveStrategy, 190
  - TransmitAndReceive, 191
- murasaki::Synchronizer, 191
  - Release, 192
  - Wait, 192
- murasaki::TaskStrategy, 192
  - GetName, 194
  - getStackDepth, 194
  - getStackMinHeadroom, 194
  - Launch, 194
  - Start, 194
  - TaskBody, 194
  - TaskStrategy, 193
- murasaki::Uart, 195
  - HandleError, 198
  - Receive, 199
  - ReceiveCompleteCallback, 199
  - SetHardwareFlowControl, 200
  - SetSpeed, 200
  - Transmit, 200
  - TransmitCompleteCallback, 202
  - Uart, 198
- murasaki::UartLogger, 203
  - DoPostMortem, 204
  - getCharacter, 204
  - putMessage, 205
- UartLogger, 204
- murasaki::UartStrategy, 205
  - HandleError, 206
  - Receive, 207
  - ReceiveCompleteCallback, 207
  - SetHardwareFlowControl, 207
  - SetSpeed, 208
  - Transmit, 208
  - TransmitCompleteCallback, 208
- murasaki\_0\_intro.hpp, 256
- murasaki\_1\_env.hpp, 256
- murasaki\_2\_ug.hpp, 257
- murasaki\_3\_pg.hpp, 257
- murasaki\_4\_mod.hpp, 257
- murasaki\_assert.hpp, 258
- murasaki\_callback.cpp, 259
  - HAL\_ADC\_ConvCpltCallback, 260
  - HAL\_ADC\_ErrorCallback, 261
  - HAL\_I2C\_MasterRxCpltCallback, 261
  - HAL\_I2C\_SlaveRxCpltCallback, 261
  - PrintFaultResult, 261
- murasaki\_config.hpp, 262
- murasaki\_defs.hpp, 263
- murasaki\_include\_stub.h, 264
- murasaki\_platform.cpp, 265
- murasaki\_platform.hpp, 266
  - MasterTaskBodyFunction, 267
  - PrintFaultResult, 267
  - SlaveTaskBodyFunction, 267
- murasaki\_syslog.hpp, 268
- Mute
  - murasaki::Adau1361, 71
  - murasaki::AudioCodecStrategy, 82
- NUM\_OF\_CALLBACK\_OBJECTS
  - Definitions and Configuration, 33
- NUM\_OF\_EXTI\_OBJECTS
  - Definitions and Configuration, 34
- operator delete
  - Helper classes, 50
- operator delete[]
  - Helper classes, 51
- operator new
  - Helper classes, 51
- operator new[]
  - Helper classes, 51
- PLATFORM\_CONFIG\_DEBUG\_BUFFER\_SIZE
  - Definitions and Configuration, 34
- PLATFORM\_CONFIG\_DEBUG\_LINE\_SIZE
  - Definitions and Configuration, 34
- PLATFORM\_CONFIG\_DEBUG\_SERIAL\_TIMEOUT
  - Definitions and Configuration, 34
- PLATFORM\_CONFIG\_DEBUG\_TASK\_PRIORITY
  - Definitions and Configuration, 34
- PLATFORM\_CONFIG\_DEBUG\_TASK\_STACK\_SIZE
  - Definitions and Configuration, 34
- peripheralstrategy.hpp, 269

- platform
  - murasaki, 68
- platform\_config.hpp, 270
  - MURASAKI\_CONFIG\_NOSYSLOG, 271
- platform\_defs.hpp, 271
- PrintFaultResult
  - murasaki\_callback.cpp, 261
  - murasaki\_platform.hpp, 267
- Printf
  - murasaki::Debugger, 102
- Put
  - murasaki::FifoStrategy, 123
- putMessage
  - murasaki::LoggerStrategy, 153
  - murasaki::UartLogger, 205
- QuadratureEncoder
  - murasaki::QuadratureEncoder, 158
- quadratureencoder.cpp, 272
- quadratureencoder.hpp, 273
- quadratureencoderstrategy.hpp, 274
- RePrint
  - murasaki::Debugger, 102
- Receive
  - murasaki::DebuggerUart, 107
  - murasaki::I2cMaster, 130
  - murasaki::I2cMasterStrategy, 134
  - murasaki::I2cSlave, 138
  - murasaki::I2cSlaveStrategy, 141
  - murasaki::Uart, 199
  - murasaki::UartStrategy, 207
- ReceiveCompleteCallback
  - murasaki::DebuggerUart, 107
  - murasaki::I2cCallbackStrategy, 125
  - murasaki::I2cMaster, 130
  - murasaki::I2cSlave, 138
  - murasaki::Uart, 199
  - murasaki::UartStrategy, 207
- Release
  - murasaki::Exti, 119
  - murasaki::InterruptStrategy, 151
  - murasaki::Synchronizer, 192
- RemoveSyslogFacilityFromMask
  - murasaki, 67
- STM32F4xx\_System\_Private\_Defines, 58
  - VECT\_TAB\_OFFSET, 58
- STM32F4xx\_System\_Private\_FunctionPrototypes, 61
- STM32F4xx\_System\_Private\_Functions, 62
  - SystemCoreClockUpdate, 62
  - SystemInit, 63
- STM32F4xx\_System\_Private\_Includes, 56
  - HSE\_VALUE, 56
  - HSI\_VALUE, 56
- STM32F4xx\_System\_Private\_Macros, 59
- STM32F4xx\_System\_Private\_TypesDefinitions, 57
- STM32F4xx\_System\_Private\_Variables, 60
- SaiPortAdapter
  - murasaki::SaiPortAdapter, 165
- saiportadapter.hpp, 276
- SendCommand
  - murasaki::Adau1361, 71
  - murasaki::AudioCodecStrategy, 82
- SendCommandTable
  - murasaki::Adau1361, 71
- Set
  - murasaki::BitOut, 95
  - murasaki::BitOutStrategy, 97
  - murasaki::QuadratureEncoder, 159
  - murasaki::QuadratureEncoderStrategy, 161
- SetAuxInputGain
  - murasaki::Adau1361, 72
- SetGain
  - murasaki::Adau1361, 72
  - murasaki::AudioCodecStrategy, 83
- SetHardwareFlowControl
  - murasaki::DebuggerUart, 108
  - murasaki::Uart, 200
  - murasaki::UartStrategy, 207
- SetHpOutputGain
  - murasaki::Adau1361, 72
- SetLineInputGain
  - murasaki::Adau1361, 72
- SetLineOutputGain
  - murasaki::Adau1361, 73
- SetPostMortem
  - murasaki::DebuggerFifo, 105
- SetSampleClock
  - murasaki::Adc, 77
  - murasaki::AdcStrategy, 81
- SetSpeed
  - murasaki::DebuggerUart, 108
  - murasaki::Uart, 200
  - murasaki::UartStrategy, 208
- SetSyslogFacilityMask
  - murasaki, 67
- SetSyslogSererityThreshold
  - murasaki, 67
- severity\_
  - murasaki::Debugger, 102
- SimpleTask
  - murasaki::SimpleTask, 170
- simpletask.hpp, 277
- SlaveTaskBodyFunction
  - murasaki\_platform.hpp, 267
- SpiClockPhase
  - Definitions and Configuration, 36
- SpiClockPolarity
  - Definitions and Configuration, 36
- SpiMaster
  - murasaki::SpiMaster, 176
- SpiSlave
  - murasaki::SpiSlave, 183
- SpiSlaveAdapter
  - murasaki::SpiSlaveAdapter, 186, 187
- SpiSlaveAdapterStrategy

