Murasaki Class Library

2.0.0

Generated by Doxygen 1.8.11

Contents

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

Muras

2 Target and Environment

Murasaki library is developed with following environment:

- `Nucleo F722ZE ( STM32F722ZE : Cortex-M7 )`
- `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
- Asyncronous
- 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, CustomDefault↩ Handler() 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 urasaki::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::ktp↩ Realtime 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 Start↩ DefaultTask() in the Src/main.c. The StartDefaultTas() 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 InitPlatoform(). 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_platfrom.cpp is compiled by C++ compiler while the main.c is compiled by C compiler. This allows the programmer to uses 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(&huart3);
    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 DebgguerUart receives the pointer to the huart3 as a parameter. The hauart3 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 US↵E_FULL_ASSERT macro inside stm32xxxx_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 CustomerAssertFailed() 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 CustomDefaultHanlder() 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-motem 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. For example, the following is the sample of a callback.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef * huart)
{
    // Poll all uart rx related interrupt receivers.
    // If hit, return. If not hit,check next.
    if (murasaki::platform.uart_console->ReceiveCompleteCallback(huart))
        return;

}
```

This callback is called from HAL, after the end of peripheral interrupt processing. In this example, programmer calls the ReceiveCompleteCallback() of the UART object in the platform from the HAL_UART_RxCpltCallback().

Note 1: Murasaki object returns true if the callback member function parameter matches with its own hardware handle.

Note 2: Forwarding this call back to all the relevant peripheral is the responsibility of the porting programmer.

```
if (murasaki::platform.uart_console->ReceiveCompleteCallback(huart))
    return;
if (murasaki::platform.uart_1->ReceiveCompleteCallback(huart))
    return;
if (murasaki::platform.uart_2->ReceiveCompleteCallback(huart))
    return;
```

3.6.6 EXTI flow

The EXTI flow is very similar to the General Interrupt flow except its timing. While other peripheral raises interrupt after the peripheral instance are created, The EXTI peripheral may raise the interrupt before the platform peripherals are ready.

Then, The EXTI call back has a guard to avoid the null pointer access.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{

    if ( USER_Btn_Pin == GPIO_Pin) {
        // release the waiting task
        if (murasaki::platform.sync_with_button != nullptr)
            murasaki::platform.sync_with_button->Release();
    }

}
```

Note that the USER_Btn_Pin constant in the above example is generated by CubeIDE, when customer labels "U↩SER_Btn" to some EXTI input pin.

Murasaki provides the murasaki::Exit class to make the waiting interrupt easy. You may want to use that rather than handling like above.

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 Custome 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.

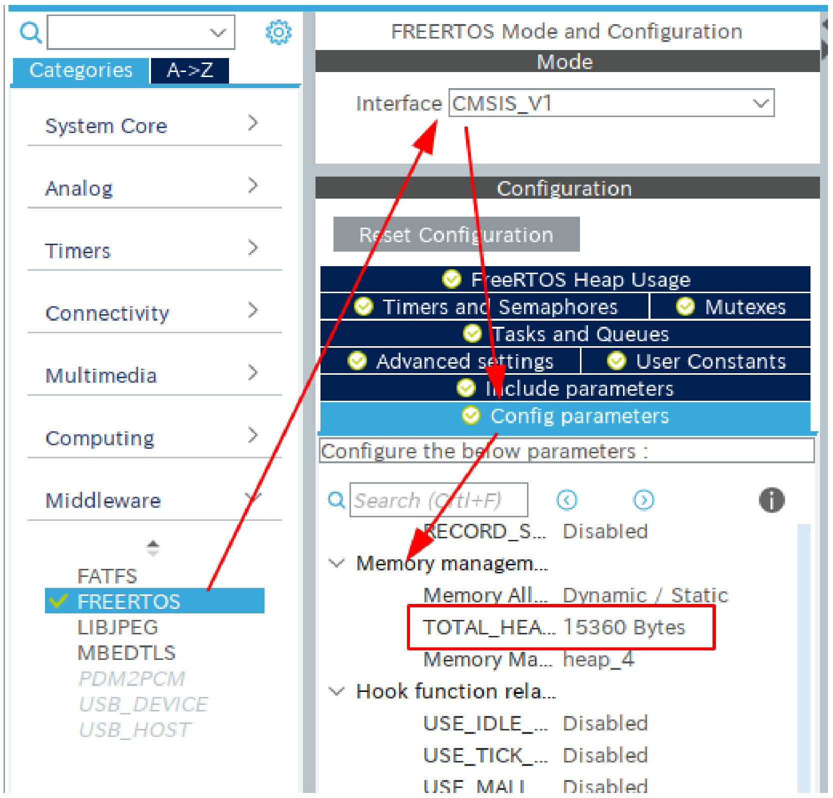
First, class instances are created inside heap region by new operator often. And second, murasaki::Debugger allocates a huge size of FIFO buffer. This buffer stays in between the murasaki::Debugger::Printf() function and the logger task. The size of this FIFO buffer is defined by PLATFORM_CONFIG_DEBUG_BUFFER_SIZE.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



Interface CMSIS_V1

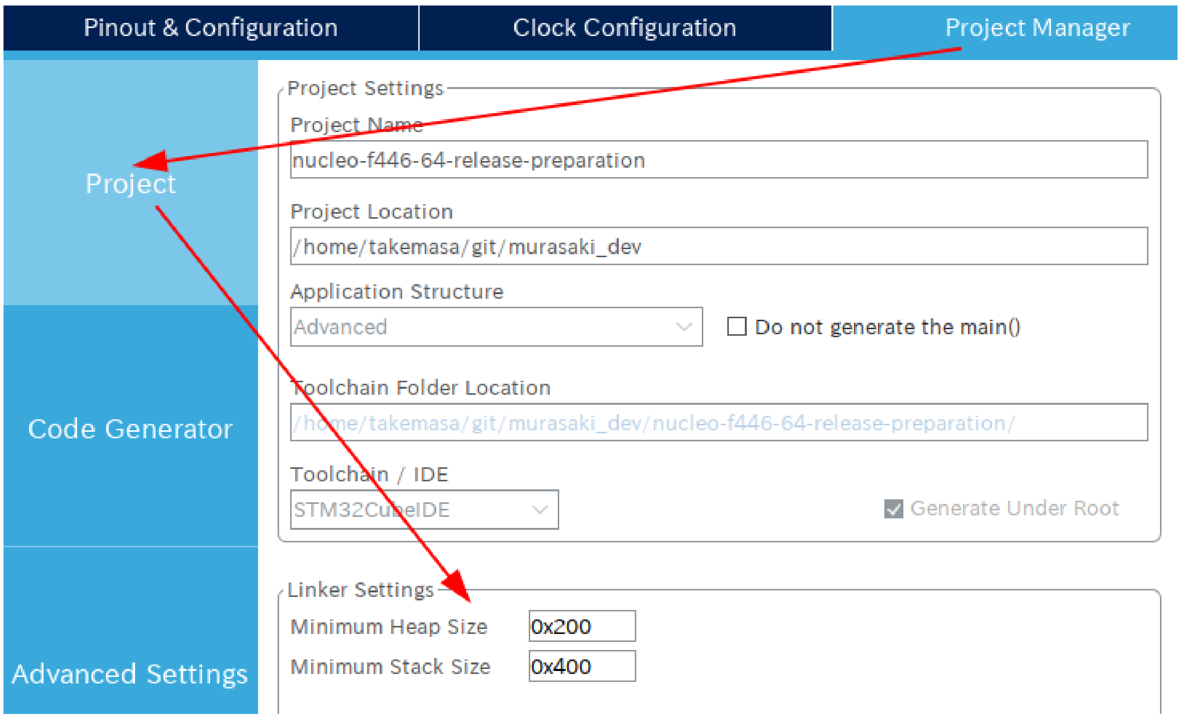
Configure the below parameters :

RECORD_S...	Disabled
Memory managem...	
Memory All...	Dynamic / Static
TOTAL_HEA...	15360 Bytes
Memory Ma...	heap_4
Hook function rela...	
USE_IDLE_...	Disabled
USE_TICK_...	Disabled
USE_MALL	Disabled

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.



Project Settings

Project Name: nucleo-f446-64-release-preparation

Project Location: /home/takemasa/git/murasaki_dev

Application Structure: Advanced

☐ Do not generate the main()

Toolchain Folder Location: /home/takemasa/git/murasaki_dev/nucleo-f446-64-release-preparation/

Toolchain / IDE: STM32CubeIDE

☑ Generate Under Root

Linker Settings

Minimum Heap Size: 0x200

Minimum Stack Size: 0x400

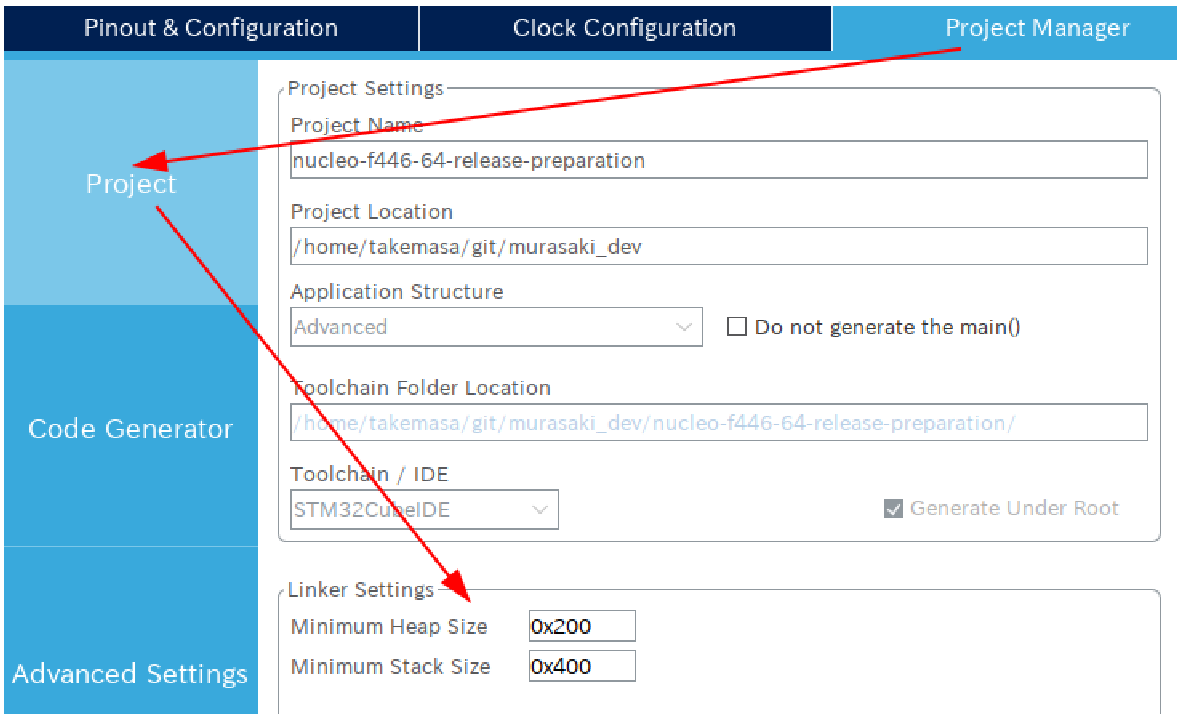
4.2.2 Stack Size

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 :



Project Settings

Project Name: nucleo-f446-64-release-preparation

Project Location: /home/takemasa/git/murasaki_dev

Application Structure: Advanced

☐ Do not generate the main()

Toolchain Folder Location: /home/takemasa/git/murasaki_dev/nucleo-f446-64-release-preparation/

Toolchain / IDE: STM32CubeIDE

☑ Generate Under Root

Linker Settings

Minimum Heap Size: 0x200

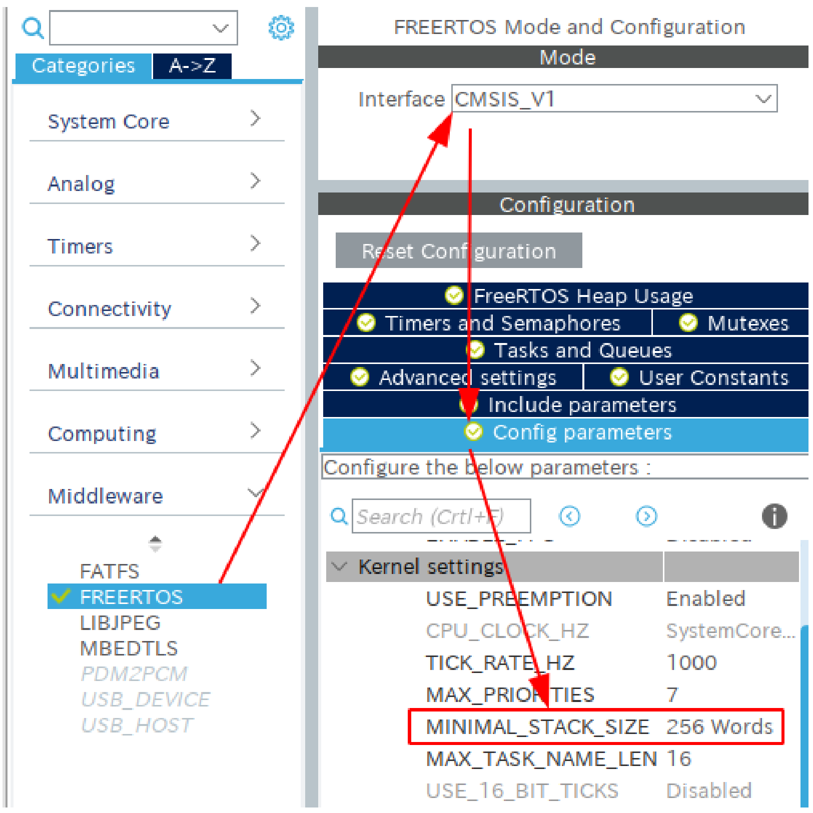
Minimum Stack Size: 0x400

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:



Interface CMSIS_V1

USE_PREEMPTION	Enabled
CPU_CLOCK_HZ	SystemCore...
TICK_RATE_HZ	1000
MAX_PRIORITIES	7
MINIMAL_STACK_SIZE	256 Words
MAX_TASK_NAME_LEN	16
USE_16_BIT_TICKS	Disabled

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_cosole 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 skeltons correctly.

In the Murasaki manner, the skeletons have to call the relevant callback member function of platform variables. For example, this is the typical programming of the call back :

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef * huart)
{
    if (murasaki::platform.uart_console->TransmitCompleteCallback(huart))
        return;

}
```

In this sample, the TxCpltCallback() calles murasaki::platform.uart_console->TransmitCompleteCallback() member funciton. And then return if that member function returns true. Note that all the callacks in the Murasaki class returns true if the given peripheral handle matches with its internal handle. Thus, this is good way to poll all the UART peripheral inside this callback function.

Following is the list of the interrupts which applicaiton have to route to the peripehral 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_stm32xxxxx.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 :

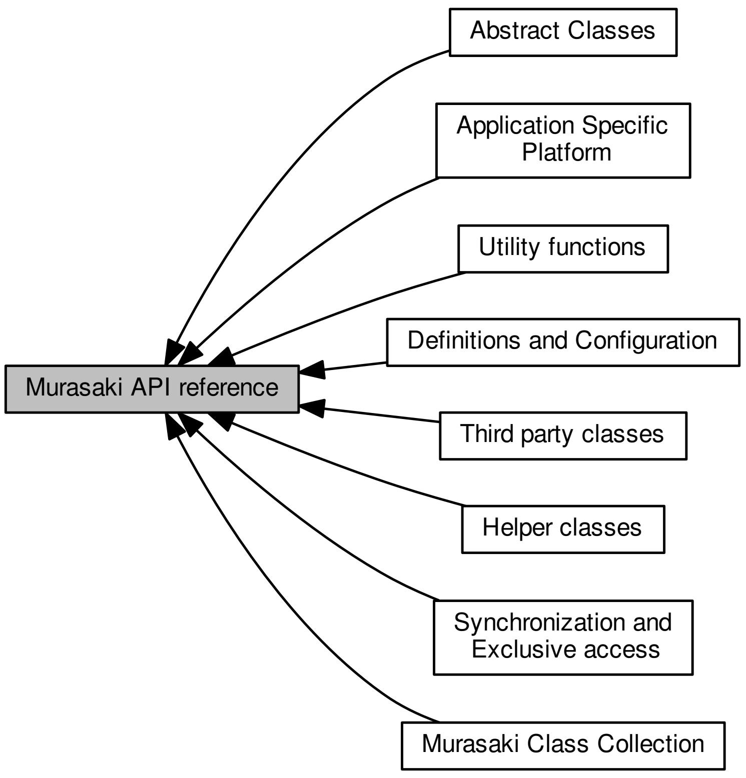
- Adjust heap size and stack size as described in the CubeIDE setting
- Generate an application skeleton from CubeIDE.
- Checkout Murasaki repository into your project.
- Copy the template files as described in the Directory Structure .
- Configure Muraaski as described in the Configuration and the Task 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:


Abstract Classes
Application Specific Platform
Utility functions
Definitions and Configuration
Murasaki API reference
Third party classes
Helper classes
Synchronization and Exclusive access
Murasaki Class Collection

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

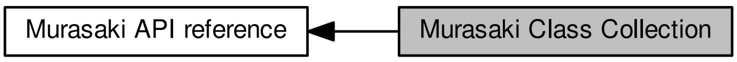
    Collection of the useful functions.

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 the descendants of 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 Free↩ RTOS.
- class murasaki::I2cSlave

  Thread safe, synchronous and blocking IO. Encapsulating I2C slave. Based on STM32Cube HAL driver and Free↩ RTOS.
- 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 Free↩ RTOS.
- class murasaki::SpiSlave

  Thread safe, synchronous and blocking IO. Encapsulating SPI slave. Based on STM32Cube HAL driver and Free↩ RTOS.
- class murasaki::SpiSlaveAdapter

  A speficier 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

COND	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.

```
{ 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

ERR	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.

```
bool Uart::HandleError(void* const ptr)
{
    MURASAKI_ASSERT(nullptr != ptr)

    if (peripheral_ == ptr) {
        // Check error, and print if exist.
        MURASAKI_PRINT_ERROR(peripheral_->ErrorCode & HAL_UART_ERROR_DMA);
        MURASAKI_PRINT_ERROR(peripheral_->ErrorCode & HAL_UART_ERROR_PE);
        MURASAKI_PRINT_ERROR(peripheral_->ErrorCode & HAL_UART_ERROR_NE);
        MURASAKI_PRINT_ERROR(peripheral_->ErrorCode & HAL_UART_ERROR_FE);
        MURASAKI_PRINT_ERROR(peripheral_->ErrorCode & HAL_UART_ERROR_ORE);
        MURASAKI_PRINT_ERROR(peripheral_->ErrorCode & HAL_UART_ERROR_DMA);
        return true;    // report the ptr matched
    }
    else {
        return false;   // report the ptr doesn't match
    }
}
```

5.2.2.3 #define MURASAKI_SYSLOG(*OBJPTR,* *FACILITY,* *SEVERITY,* *FORMAT,* ...)

Parameters

OBJPTR	the pointer to the object. Usually, path the "this" pointer here.
FACILITY	Specify which facility makes this log. Choose from murasaki::SyslogFacility
SEVERITY	Specify how message is severe. Choose from murasaki::SyslogSeverity
FORMAT	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 falty condtion from HAL or hardware.
- murasaki::kseEmergency for software logic error like assert fail

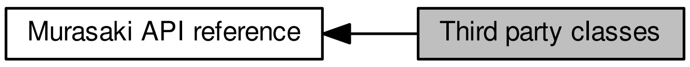
The output format is as following :

- 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

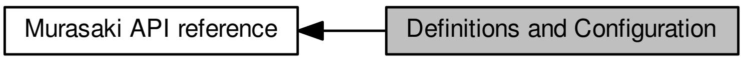
- class murasaki::Adau1361
  Audio Codec LSI class.

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 proiority of the debug task.
- #define MURASAKI_CONFIG_NODEBUG false

  Suppress MURASAKI_ASSERT macro.
- #define MURASAKI_CONFIG_NOCYCCNT false

  Doesn't run the CYCCNT counter.

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 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.

5.4.2.4 #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.

5.4.2.5 #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.

5.4.2.6 #define PLATFORM_CONFIG_DEBUG_TASK_PRIORITY murasaki::ktpHigh

The priority of the murasaki::Debuger internal task. To output the logging data as fast as possible, the debug taks 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.

5.4.2.7 #define PLATFORM_CONFIG_DEBUG_TASK_STACK_SIZE 256

The murasaki::Debuger 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 answeres 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 corrision.
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

kutNoIdleTimeout	kutNoIdleTimeout 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 milli-seconds. 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.

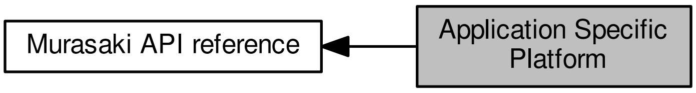
Enumerator

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 ∗hspi)

  Essential to sync up with SPI.
- void HAL_SPI_ErrorCallback (SPI_HandleTypeDef ∗hspi)

  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_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 platfrom.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::Uart↩ Logger 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

file	Name of the source file where assertion happen
line	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.

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

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

5.5.2.2 void CustomDefaultHandler ( )

An entry of the exception. Especialy 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 leve, when you investigate the Hard Fault.

For example, the start up code for the Nucleo-L152RE is startup_stml152xe.s. This file is generated by CubeIDE. This file has default handler as like this:

```
.section .text.Default_Handler,"ax",%progbits
    Default_Handler:
Infinite_Loop:
    b Infinite_Loop
```

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

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

GPIO_Pin	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

hi2c	

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

hi2c	

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

hi2c	

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_SAI_ErrorCallback (SAI_HandleTypeDef ∗ *hsai*)

Parameters

hsai	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.

5.5.2.9 void HAL_SAI_RxCpltCallback (SAI_HandleTypeDef ∗ *hsai*)

Parameters

hsai	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.

5.5.2.10 void HAL_SAI_RxHalfCpltCallback (SAI_HandleTypeDef ∗ *hsai*)

Parameters

hsai	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.

5.5.2.11 void HAL_SPI_ErrorCallback (SPI_HandleTypeDef ∗ *hspi*)

Parameters

hspi	

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.

5.5.2.12 void HAL_SPI_TxRxCpltCallback (SPI_HandleTypeDef ∗ *hspi*)

Parameters

hspi	

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::TransmitAndReceiveComplete↩ Callback () function.

5.5.2.13 void HAL_UART_ErrorCallback (UART_HandleTypeDef ∗ *huart*)

Parameters

huart	

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.

5.5.2.14 void HAL_UART_RxCpltCallback (UART_HandleTypeDef ∗ *huart*)

Parameters

huart	

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.

5.5.2.15 void HAL_UART_TxCpltCallback (UART_HandleTypeDef ∗ *huart*)

Parameters

huart	

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.

5.5.2.16 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.

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

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

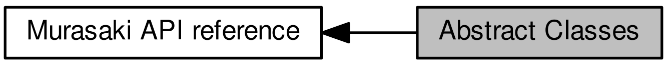
5.5.3.1 murasaki::Debugger ∗ murasaki::debugger

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::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::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.

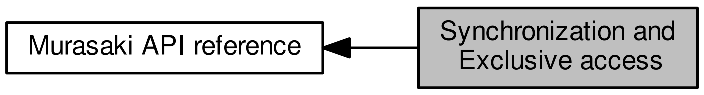
5.6.1 Detailed Description

Usually, application dodesn't instantiate these classes. But pointer may be declared as abstract class as geneic 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.

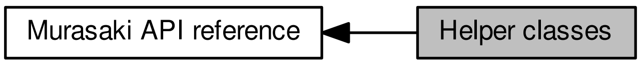
5.7.1 Detailed Description

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::DebuggerFifo

    FIFO with thread safe.
- struct murasaki::LoggingHelpers

    A stracture to engroup the logging tools.
- class murasaki::DebuggerUart

    Logging dedicated UART class.

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

ptr	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

ptr	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

size	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

size	Size of the memory to allocate [byte]

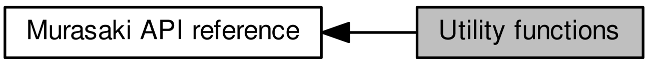
Returns

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 serach 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 noting in this function.

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

5.9.2.2 void murasaki::I2cSearch (murasaki::I2CMasterStrategy ∗ *master*)

Parameters

master	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 ( )

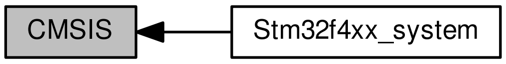
This cycle counter (CYCNT) is implemented inside CORTEX-Mx core. To implement or not is up to the SoC vender.

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:



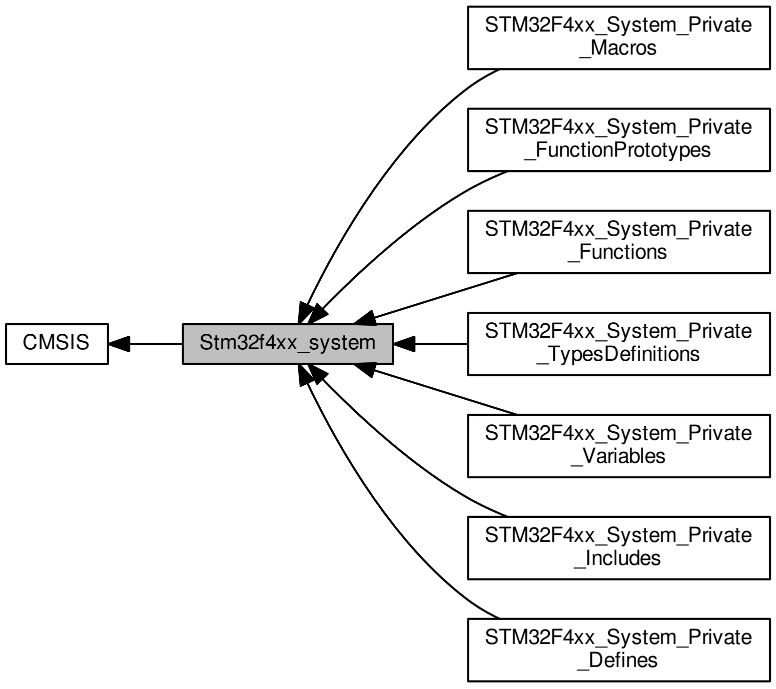
Modules

- Stm32f4xx_system

5.10.1 Detailed Description

5.11 Stm32f4xx_system

Collaboration diagram for Stm32f4xx_system:


STM32F4xx_System_Private _Macros
STM32F4xx_System_Private _FunctionPrototypes
STM32F4xx_System_Private _Functions
CMSIS
Stm32f4xx_system
STM32F4xx_System_Private _TypesDefinitions
STM32F4xx_System_Private _Variables
STM32F4xx_System_Private _Includes
STM32F4xx_System_Private _Defines

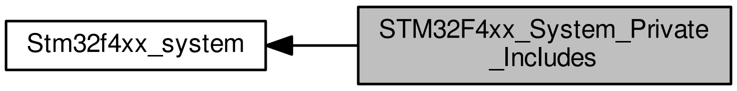
Modules

- 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)

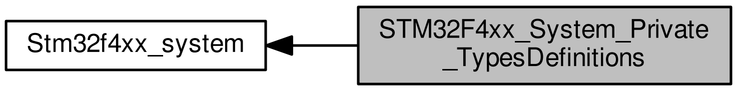
Default value of the External oscillator in Hz

5.12.2.2 #define HSI_VALUE ((uint32_t)16000000)

Value of the Internal oscillator in Hz

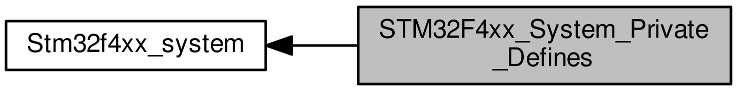
5.13 STM32F4xx_System_Private_TypesDefinitions

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

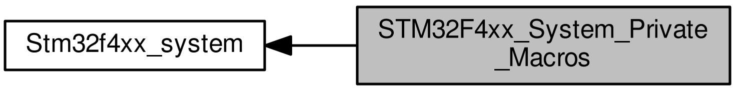
5.14.2.1 #define VECT_TAB_OFFSET 0x00

< 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.

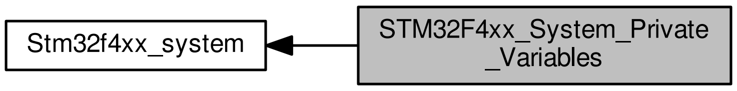
5.15 STM32F4xx_System_Private_Macros

Collaboration diagram for STM32F4xx_System_Private_Macros:



5.16 STM32F4xx_System_Private_Variables

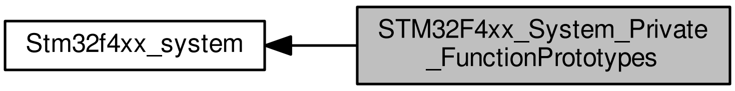
Collaboration diagram for STM32F4xx_System_Private_Variables:



5.16.1 Detailed Description

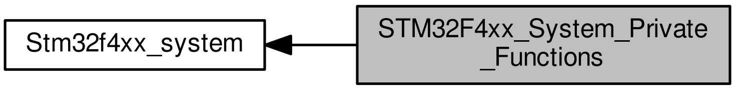
5.17 STM32F4xx_System_Private_FunctionPrototypes

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

None	

Return values

None	

5.18.2.2 void SystemInit (void)

Parameters

None	

Return values

None	

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 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 the descendants of this class.
- class Exti

  EXTI wrapper class.
- class FifoStrategy

  Basic FIFO without thread safe.
- struct GPIO_type

  A structure to en-group the GPIO port and GPIO pin.
- class I2cMaster

  Thread safe, synchronous, and blocking IO. Encapsulating I2C master. Based on STM32Cube HAL driver and Free↩ RTOS.
- 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 Free↩ RTOS.

- 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 stracture 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 SpiMaster

  Thread safe, synchronous and blocking IO. Encapsulating SPI master. Based on STM32Cube HAL driver and Free←↩ RTOS.
- 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 Free←↩ RTOS.
- class SpiSlaveAdapter

  A speficier 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

facility	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

facility	Message facility
severity	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

facility	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

mask	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

severity	

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.

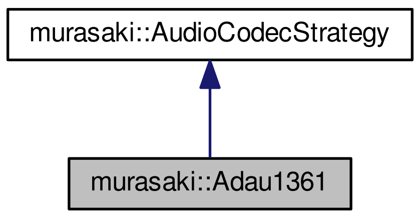
7 Class Documentation

7.1 murasaki::Adau1361 Class Reference

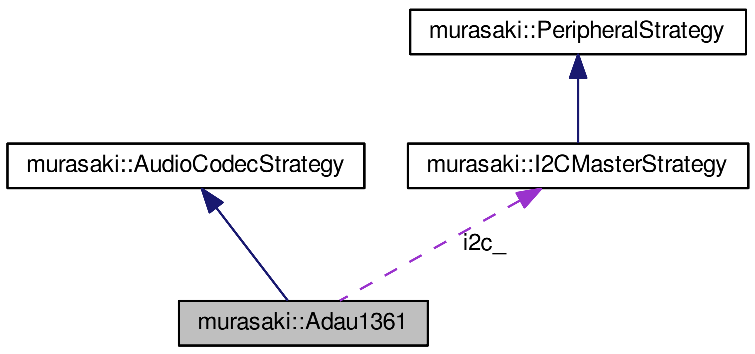
Audio Codec LSI class.

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

Inheritance diagram for murasaki::Adau1361:



Collaboration diagram for murasaki::Adau1361:


murasaki::PeripheralStrategy
murasaki::AudioCodecStrategy
murasaki::I2CMasterStrategy
i2c_
murasaki::Adau1361

Public Member Functions

- Adau1361 (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

fs	Sampling frequency[Hz]
master_clock	Input master clock frequency to the MCLK pin[Hz]
controller	Pass the I2C controller object.
i2c_device_addr	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

channel	Channel to mute on / off
mute	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

command	command data array. It have to have register addess of ADAU1361 in first two bytes.
size	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

table	command table. All commands are stored in one row. Each row has only 1 byte data after reg address.
rows	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

left_gain	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
right_gain	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
mute	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

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

left_gain	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
right_gain	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
mute	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

left_gain	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
right_gain	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
mute	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

left_gain	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
right_gain	Gain by dB. [6 .. -12], The gain value outside of the acceptable range will be saturated.
mute	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:

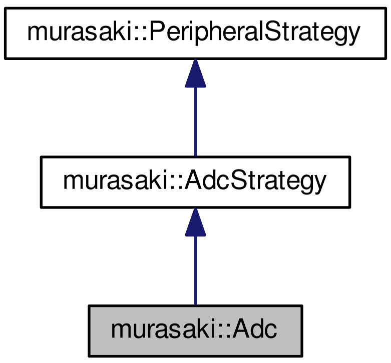
- adau1361.hpp

7.2 murasaki::Adc Class Reference

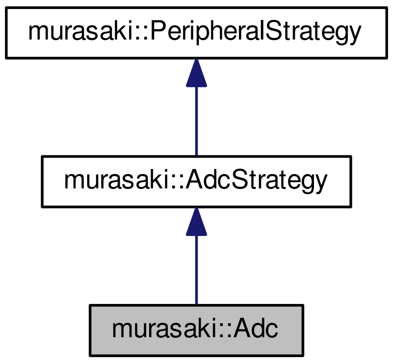
STM32 dedicated ADC class.

`#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 ()

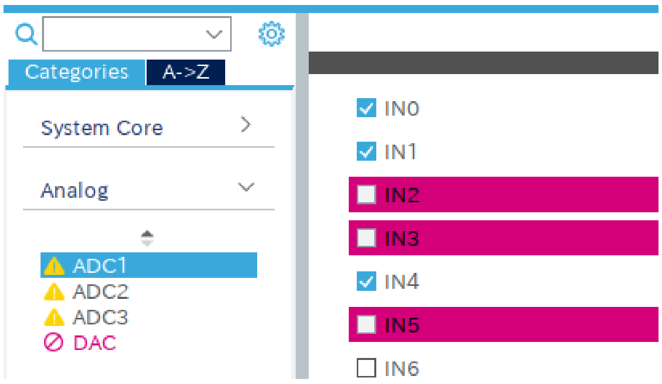
  pass the raw peripheral handler

7.2.1 Detailed Description

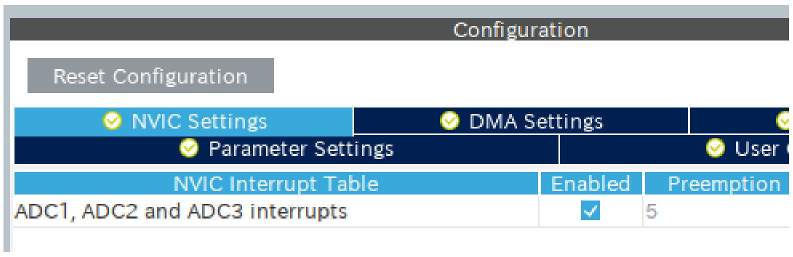
This is a single conversion ADC class. That mean, Only one specified channel will be converted at onces and then, return. Optionally, The sampling clock duration can be set for each channels.

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.



NVIC Interrupt Table	Enabled	Preemption
ADC1, ADC2 and ADC3 interrupts	✓	5

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

The interrupt handler is the HAL_ADC_ConvCpltCallback() and the HAL_ADC_ErrorCallback(). These are regulated by HAL manual. If you define these functions, HAL will call that callback whenever the relevant interrupt happens.

Inside HAL_ADC_ConvCpltCallback(), you must call the Adc::ConversionCompleteCallback() member function. The parameter should be the hadc parameter of the Interrupt callback. The Adc::ConversionCompleteCallback() will return with if the given hadc is the handle of that object. If the Adc::ConversionCompleteCallback() return true, you can return from the handler.

The HAL_ADC_ErrorCallback() is similar but you must call Adc::HandleError().

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

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

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

peripheral	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

ptr	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

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

Returns

Status. murasaki::kasOk is return if success.

Convert the analog singal 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

ptr	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

channel	Specify the ADC channel.Use ADC_CHANNEL_∗ value.
clocks	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:

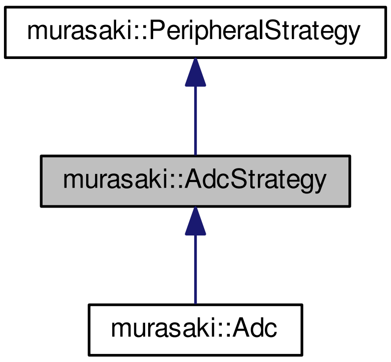
- adc.hpp
- adc.cpp

7.3 murasaki::AdcStrategy Class Reference

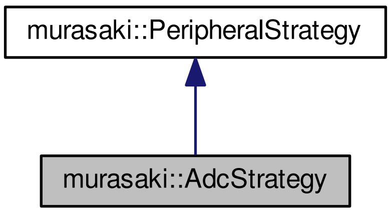
Synchronized, blocking ADC converter Strategy.

```
#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

ptr	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

channel	Specify the ADC channel.
value	Pointer to the variable to receive the data. Data is normalized by range [-1,1)
size	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

ptr	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

channel	Specify the ADC channel.
clocks	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:

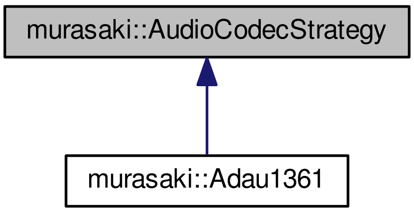
- adcstrategy.hpp

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

fs	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

channel	Channel to mute on / off
mute	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

command	command data array.
size	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

channel	
left_gain	[dB]
right_gain	[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:

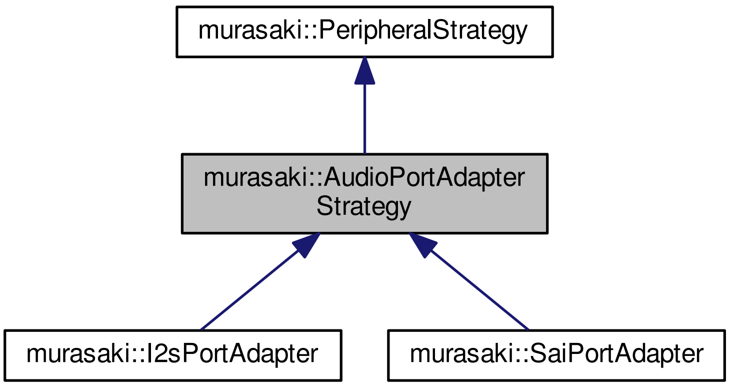
- audiocodecstrategy.hpp

7.5 murasaki::AudioPortAdapterStrategy Class Reference

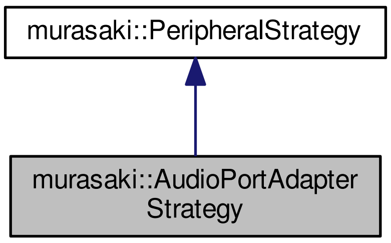
Strategy of the audio device adaptor..

`#include <audioportadapterstrategy.hpp>`

Inheritance diagram for murasaki::AudioPortAdapterStrategy:


murasaki::PeripheralStrategy
murasaki::AudioPortAdapter Strategy
murasaki::I2sPortAdapter
murasaki::SaiPortAdapter

Collaboration diagram for murasaki::AudioPortAdapterStrategy:


murasaki::PeripheralStrategy
murasaki::AudioPortAdapter Strategy

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

phase	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

ptr	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 :harlf 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

peripheral_handle	

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:

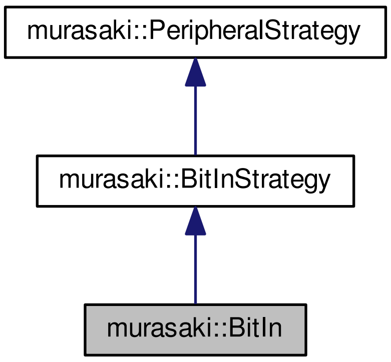
- audioportadapterstrategy.hpp

7.6 murasaki::BitIn Class Reference

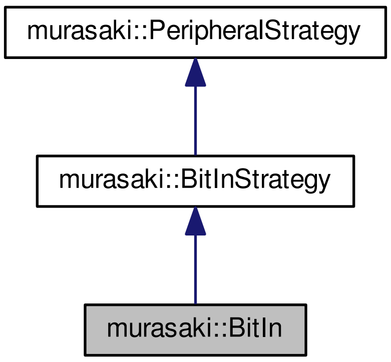
General purpose bit input.

`#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

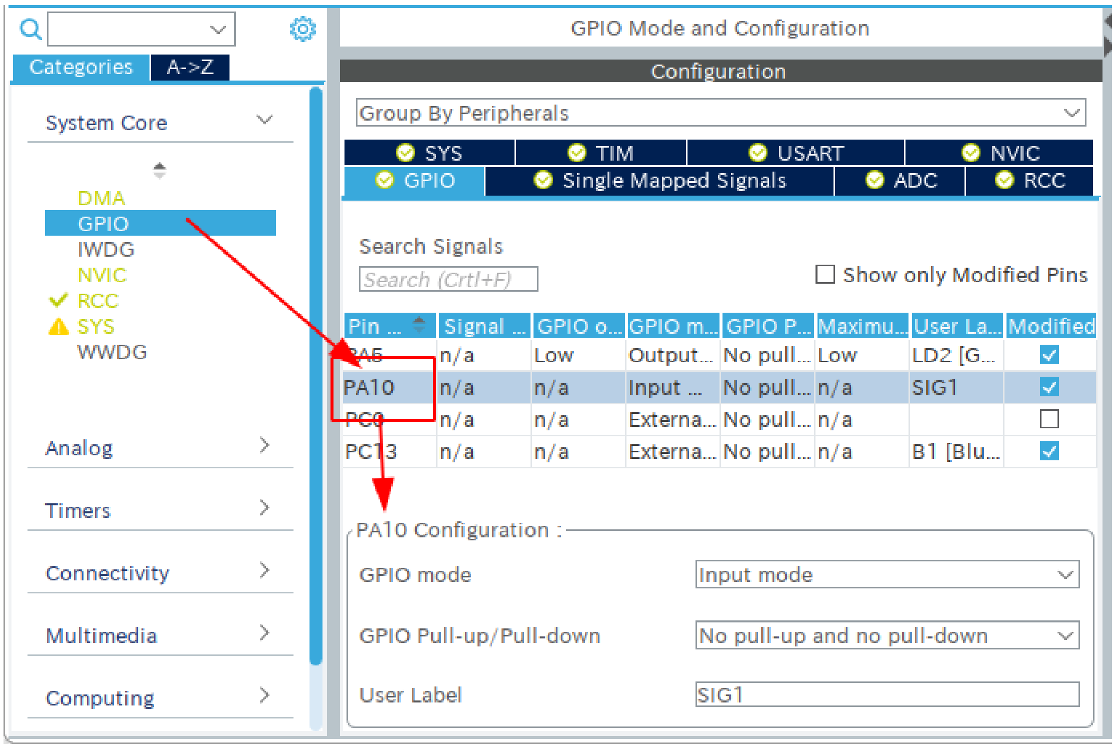
7.6.1 Detailed Description

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"



Pin ...	Signal ...	GPIO o...	GPIO m...	GPIO P...	Maximu...	User La...	Modified
PA5	n/a	Low	Output...	No pull...	Low	LD2 [G...	✓
PA10	n/a	n/a	Input ...	No pull...	n/a	SIG1	✓
PC0	n/a	n/a	Externa...	No pull...	n/a		☐
PC13	n/a	n/a	Externa...	No pull...	n/a	B1 [Blu...	✓

PA10 Configuration :

GPIO mode	Input mode
GPIO Pull-up/Pull-down	No pull-up and no pull-down
User Label	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::BitIn (GPIO_TypeDef ∗ *port*, uint16_t *pin*)

Parameters

port	Pinter to the port strict.
pin	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:

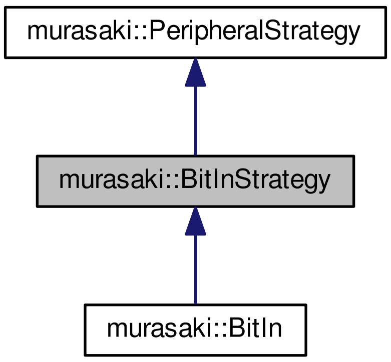
- bitin.hpp
- bitin.cpp

7.7 murasaki::BitInStrategy Class Reference

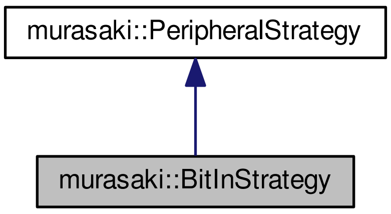
Definition of the root class of bit input.

```
#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:

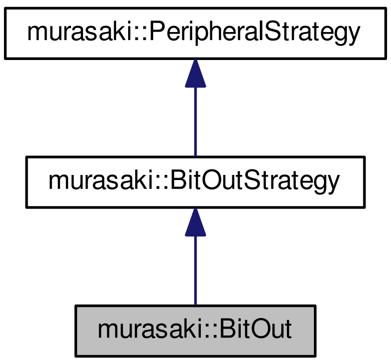
- bitinstrategy.hpp

7.8 murasaki::BitOut Class Reference

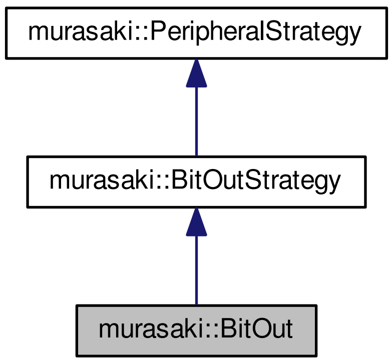
General purpose bit output.

```
#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

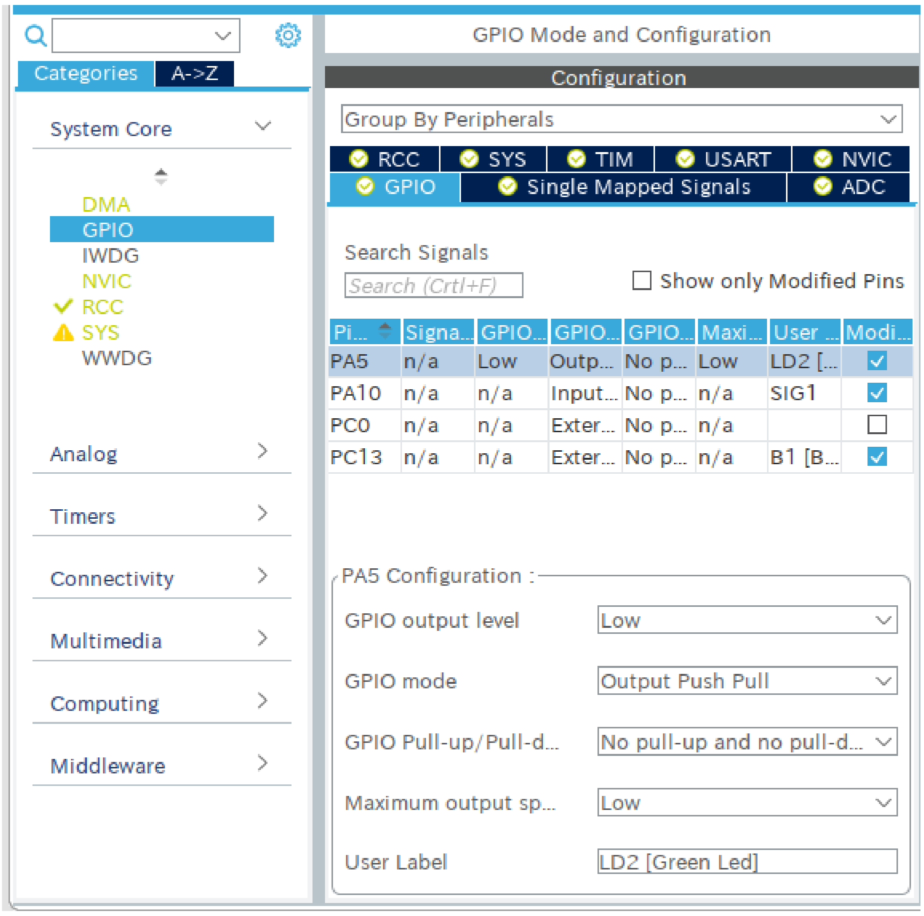
7.8.1 Detailed Description

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"



Pi...	Signa...	GPIO...	GPIO...	GPIO...	Maxi...	User ...	Modi...
PA5	n/a	Low	Outp...	No p...	Low	LD2 [...	✓
PA10	n/a	n/a	Input...	No p...	n/a	SIG1	✓
PC0	n/a	n/a	Exter...	No p...	n/a		☐
PC13	n/a	n/a	Exter...	No p...	n/a	B1 [B...	✓

PA5 Configuration :

GPIO output level	Low
GPIO mode	Output Push Pull
GPIO Pull-up/Pull-d...	No pull-up and no pull-d...
Maximum output sp...	Low
User Label	LD2 [Green Led]

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

port	Pinter to the port strict.
pin	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

state	Set "H" if the value is none zero, vice versa.

Implements murasaki::BitOutStrategy.

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

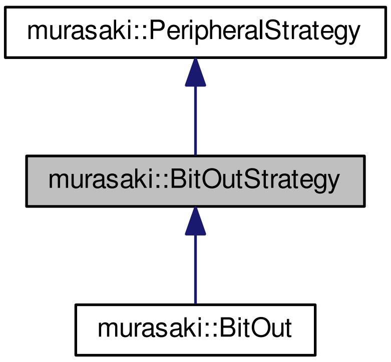
- bitout.hpp
- bitout.cpp

7.9 murasaki::BitOutStrategy Class Reference

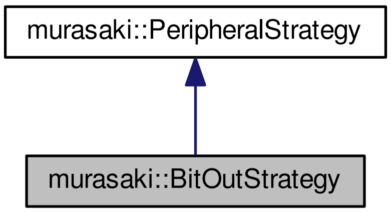
Definition of the root class of bit output.

```
#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

state	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::CriticalSection Class Reference

A critical section for task context.

```
#include <criticalsection.hpp>
```

Public Member Functions

- CriticalSection ()

  Constructor. Creating semaphore internally.
- virtual ∼CriticalSection ()

  Destructor. Deleting sempahore internally.
- void Enter ()

  Entering critical section.
- void Leave ()

  Leaving crititical section.

7.10.1 Detailed Description

The critical section prevent other task to preempt that critical section. So, a task can modify the shared variable safely inside critical seciton.

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().

7.10.2 Member Function Documentation

7.10.2.1 void murasaki::CriticalSection::Enter ( )

Entering critical section in task context. No other task can preemptive the task inside critical section.

7.10.2.2 void murasaki::CriticalSection::Leave ( )

All critical seciton started by CriticalSection::Enter() have to be quit by this member function.

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

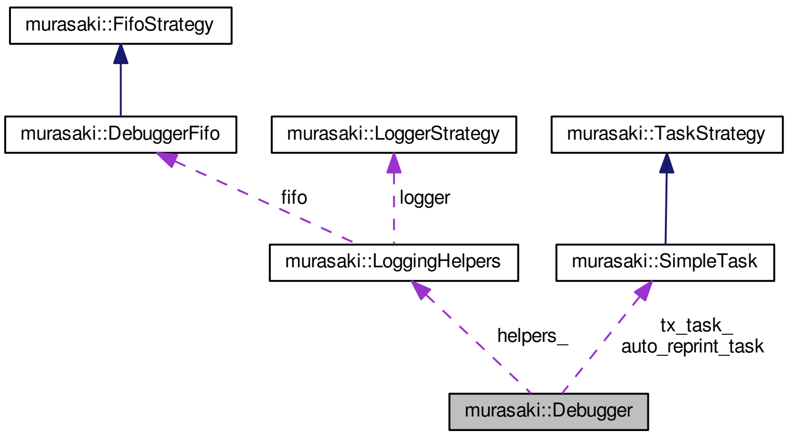
- criticalsection.hpp
- criticalsection.cpp

7.11 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 ()
  Deconstructor. 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.

7.11.1 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.

7.11.2 Constructor & Destructor Documentation

7.11.2.1 murasaki::Debugger::Debugger (LoggerStrategy ∗ *logger*)

Parameters

logger	The pointer to the LoggerStrategy wrapper class variable.

7.11.3 Member Function Documentation

7.11.3.1 void murasaki::Debugger::AutoRePrint ( )

Once this member function is called, internally new task is created. This new task watches input by GetchFrom↩ Task() 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.

7.11.3.2 char murasaki::Debugger::GetchFromTask ( )

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.

7.11.3.3 void murasaki::Debugger::Printf (const char ∗ *fmt,* ...)

Parameters

fmt	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 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.

7.11.3.4 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 .

7.11.4 Member Data Documentation

7.11.4.1 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))

7.11.4.2 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.

7.11.4.3 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:

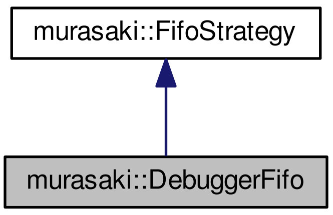
- debugger.hpp
- debugger.cpp

7.12 murasaki::DebuggerFifo Class Reference

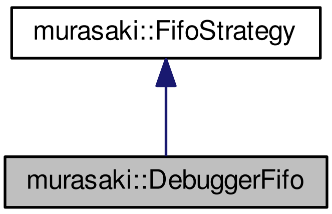
FIFO with thread safe.

```
#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

7.12.1 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.

7.12.2 Constructor & Destructor Documentation

7.12.2.1 murasaki::DebuggerFifo::DebuggerFifo (unsigned int *buffer_size*)

Parameters

buffer_size	Size of the internal buffer to be allocated [byte]

Allocate an internal buffer with given buffer_size. The buffer contents is initialized by blank.

7.12.3 Member Function Documentation

7.12.3.1 unsigned int murasaki::DebuggerFifo::Get (uint8_t *data[ ]*, unsigned int *size*) `[virtual]`

Parameters

data	Data buffer to receive from the internal buffer
size	Size of the data parameter.

Returns

The count of copied data. 0, if the internal buffer is empty

Reimplemented from murasaki::FifoStrategy.

7.12.3.2 void murasaki::DebuggerFifo::SetPostMortem ( ) `[virtual]`

In this mode, FIFO doesn't sync between the put and get method. Actually, this mode assumes nobody send messayge by Put()

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

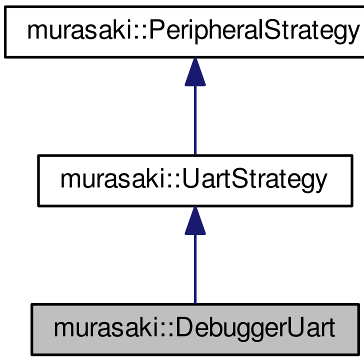
- debuggerfifo.hpp
- debuggerfifo.cpp

7.13 murasaki::DebuggerUart Class Reference

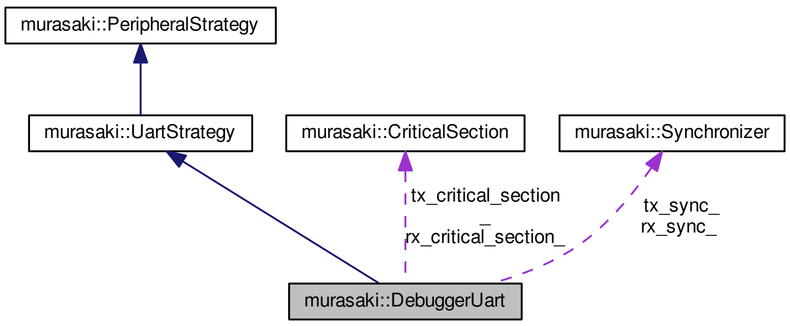
Logging dedicated UART class.

`#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 ∗transfered_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

7.13.1 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.

In addition to the instantiation, we need to prepare an interrupt callback.

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef * huart)
{
    my_uart3->TransmitCompleteCallback(huart);
}
```

Where HAL_UART_TxCpltCallback is a predefined name of the UART interrupt handler. This is invoked by system whenever a DMA baed UART transmission is complete. Becuase the default function is weakly bound, above definition will overwride the default one.

Note that above callback is invoked for any UARTn where n is 1, 2, 3... To avoid the confusion, Uart::Transmit↩CompleteCallback() method chckes whether given parameter matches with its UART_HandleTypeDef ∗ pointer (which was passed to constructor). And only when both matches, the member function execute the interrupt termination process.

As same as Tx, RX needs HAL_UART_TxCpltCallback().

Once the instance and callbacks are correctly prepared, we can use the Tx/Rx member function.

The Uart::Transmit() member function is a synchrnous function. A programmer can specify the timeout by timeout↩_ms parameter. By default, this parameter is set by kwmsIndefinitely which specifes 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 specifes 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.

7.13.2 Constructor & Destructor Documentation

7.13.2.1 murasaki::DebuggerUart::DebuggerUart (UART_HandleTypeDef ∗ *uart*)

Parameters

uart	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.

7.13.3 Member Function Documentation

7.13.3.1 bool murasaki::DebuggerUart::HandleError (void ∗const *ptr*) [virtual]

Parameters

ptr	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.

7.13.3.2 murasaki::UartStatus murasaki::DebuggerUart::Receive (uint8_t ∗ *data,* unsigned int *count,* unsigned int ∗ *transfered_count,* UartTimeout *uart_timeout,* unsigned int *timeout_ms*) [virtual]

Parameters

data	Data buffer to place the received data..
count	The count of the data (byte) to be transfered. Must be smaller than 65536
transfered_count	This parameter is ignored.
uart_timeout	This parameter is ignored
timeout_ms	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 forbiddedn to call from ISR.

Implements murasaki::UartStrategy.

7.13.3.3 bool murasaki::DebuggerUart::ReceiveCompleteCallback (void *const *ptr*) [virtual]

Parameters

ptr	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.

7.13.3.4 void murasaki::DebuggerUart::SetHardwareFlowControl (UartHardwareFlowControl *control*) [virtual]

Parameters

control	The control mode.

Before calling this method, all transmission and recevie activites have to be finished. This is responsibility of the programmer.

Note this method is NOT re-etnrant. In other word, this member function can be called from both task and interrupt context.

Reimplemented from murasaki::UartStrategy.

7.13.3.5 void murasaki::DebuggerUart::SetSpeed (unsigned int *baud_rate*) [virtual]

Parameters

baud_rate	BAUD rate (110, 300,... 57600,...)

Before calling this method, all transmission and recevie activites have to be finished. This is responsibility of the programmer.

Note this method is NOT re-etnrant. In other word, this member function can be called from both task and interrupt context.

Reimplemented from murasaki::UartStrategy.

7.13.3.6 murasaki::UartStatus murasaki::DebuggerUart::Transmit (const uint8_t * *data,* unsigned int *size,* unsigned int *timeout_ms*) [virtual]

Parameters

data	Data buffer to be transmitted.
size	The count of the data (byte) to be transfered. Must be smaller than 65536
timeout_ms	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 forbiddedn to call from ISR.

Implements murasaki::UartStrategy.

7.13.3.7 bool murasaki::DebuggerUart::TransmitCompleteCallback (void ∗const *ptr*) [virtual]

Parameters

ptr	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

7.14 murasaki::DuplexAudio Class Reference

Stereo Audio is served by the descendants of this class.

`#include <duplexaudio.hpp>`

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.

7.14.1 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.

7.14.2 Constructor & Destructor Documentation

7.14.2.1 murasaki::DuplexAudio::DuplexAudio (murasaki::AudioPortAdapterStrategy ∗ *peripheral_adapter,* unsigned int *channel_length*)

Parameters

peripheral_adapter	Pointer to the audio interface peripheral class
channel_length	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.

7.14.3 Member Function Documentation

7.14.3.1 bool murasaki::DuplexAudio::DmaCallback (void ∗ *peripheral,* unsigned int *phase*)

Parameters

peripheral	pointer to the peripheral device.
phase	0 or 1, ..., numPhase-1. The index of the buffer in the muli-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 hander.

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;
}
```

7.14.3.2 bool murasaki::DuplexAudio::HandleError (void ∗ *peripheral*) [virtual]

Parameters

peripheral	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;
}
```

7.14.3.3 void murasaki::DuplexAudio::TransmitAndReceive (float ∗ *tx_left,* float ∗ *tx_right,* float ∗ *rx_left,* float ∗ *rx_right*)

Parameters

tx_left	Pointer to the left channel TX buffer
tx_right	Pointer to the right channel TX buffer
rx_left	Pointer to the left channel RX buffer
rx_right	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
    ...
}
```

7.14.3.4 void murasaki::DuplexAudio::TransmitAndReceive (float ** *tx_channels,* float ** *rx_channels,* unsigned int *tx_num_of_channels,* unsigned int *rx_num_of_channels*)

Parameters

tx_channels	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.
rx_channels	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.
tx_num_of_channels	Any number which is smaller than or equal to num_of_channels given audio peripheral adapter.
rx_num_of_channels	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_channels_array
    ...
}
```

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

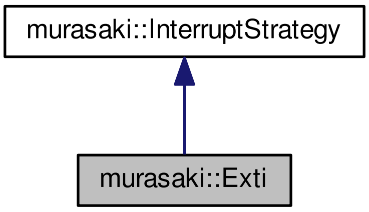
- duplexaudio.hpp
- duplexaudio.cpp

7.15 murasaki::Exti Class Reference

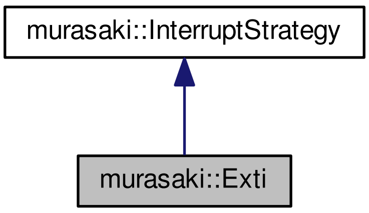
EXTI wrapper class.

`#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)

7.15.1 Detailed Description

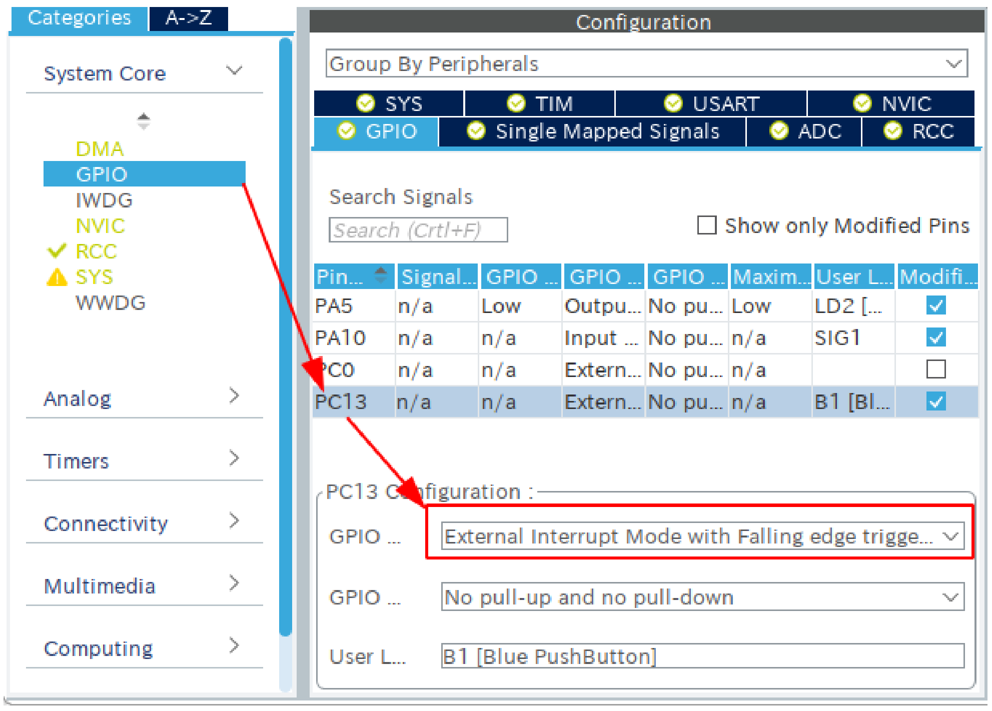
This class allows enabling/disabling EXTI interrupt from GPIO. Only EXTI_0 to EXTI_16 is supported.

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.



Pin...	Signal...	GPIO ...	GPIO ...	GPIO ...	Maxim...	User L...	Modifi...
PA5	n/a	Low	Outpu...	No pu...	Low	LD2 [...	✓
PA10	n/a	n/a	Input ...	No pu...	n/a	SIG1	✓
PC0	n/a	n/a	Extern...	No pu...	n/a		☐
PC13	n/a	n/a	Extern...	No pu...	n/a	B1 [Bl...	✓

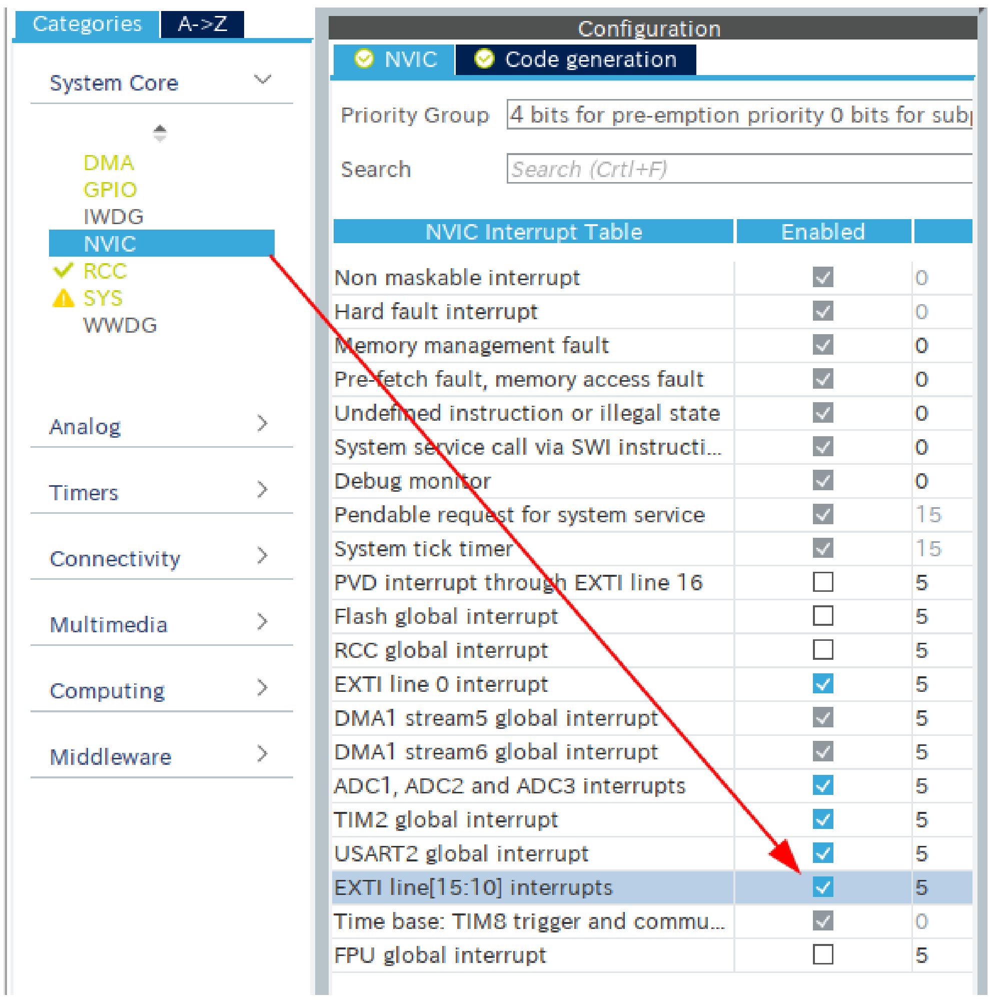
PC13 Configuration :

GPIO ... External Interrupt Mode with Falling edge trigge...

GPIO ... No pull-up and no pull-down

User L... B1 [Blue PushButton]

Next, you must enable the relevant NVIC interrupt line. In this example, we enable the NIVC of EXTI15:10 (for EXTI 13).



Priority Group: 4 bits for pre-emption priority 0 bits for subp

NVIC Interrupt Table	Enabled	
Non maskable interrupt	✓	0
Hard fault interrupt	✓	0
Memory management fault	✓	0
Pre-fetch fault, memory access fault	✓	0
Undefined instruction or illegal state	✓	0
System service call via SWI instructi...	✓	0
Debug monitor	✓	0
Pendable request for system service	✓	15
System tick timer	✓	15
PVD interrupt through EXTI line 16	☐	5
Flash global interrupt	☐	5
RCC global interrupt	☐	5
EXTI line 0 interrupt	✓	5
DMA1 stream5 global interrupt	✓	5
DMA1 stream6 global interrupt	✓	5
ADC1, ADC2 and ADC3 interrupts	✓	5
TIM2 global interrupt	✓	5
USART2 global interrupt	✓	5
EXTI line[15:10] interrupts	✓	5
Time base: TIM8 trigger and commu...	✓	0
FPU global interrupt	☐	5

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

In the interrupt callback, you can release the waiting task by calling Exti::Release(). The parameter of the HAL_↩ GPIO_EXTI_Callback() must be passed to the Release() member function. If the given parameter is the same with its EXTI line, the Exti.Release() function releases the waiting task and returns with true.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    // Check whether appropriate interrupt or not
    if (my_exti.Release(GPIO_Pin))
        return;
}
```

7.15.2 Constructor & Destructor Documentation

7.15.2.1 murasaki::Exti::Exti (unsigned int *line*)

Parameters

line	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);
```

7.15.3 Member Function Documentation

7.15.3.1 bool murasaki::Exti::Release (unsigned int *line*) [virtual]

Release the waiting task

Parameters

line	Interrupt line bit map given from the HAL_GPIO_EXTI_Callback()

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.

7.15.3.2 murasaki::InterruptStatus murasaki::Exti::Wait (unsigned int *timeout* = murasaki::kwmsIndefinitely) [virtual]

Wait for an interrupt from task.

Parameters

timeout	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:

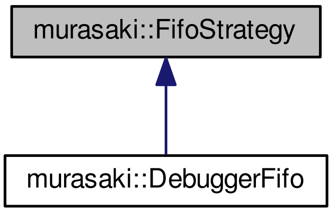
- exti.hpp
- exti.cpp

7.16 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.

7.16.1 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.

7.16.2 Constructor & Destructor Documentation

7.16.2.1 murasaki::FifoStrategy::FifoStrategy (unsigned int *buffer_size*)

Parameters

buffer_size	Size of the internal buffer to be allocated [byte]

Allocate the internal buffer with given buffer_size. The contents is not initialized.

7.16.3 Member Function Documentation

7.16.3.1 unsigned int murasaki::FifoStrategy::Get (uint8_t *data[ ],* unsigned int *size*) `[virtual]`

Parameters

data	Data buffer to receive from the internal buffer
size	Size of the data parameter.

Returns

The count of copied data. 0, if the internal buffer is empty

Reimplemented in murasaki::DebuggerFifo.

7.16.3.2 unsigned int murasaki::FifoStrategy::Put (uint8_t const *data[ ],* unsigned int *size*) `[virtual]`

Parameters

data	Data to be copied to the internal buffer
size	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

7.17 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.

7.17.1 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:

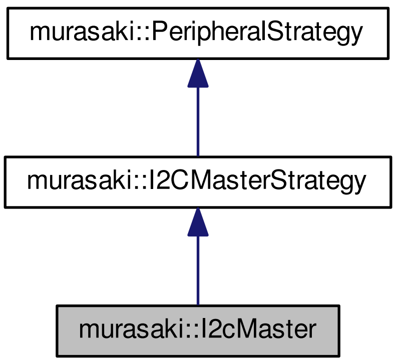
- bitout.hpp

7.18 murasaki::I2cMaster Class Reference

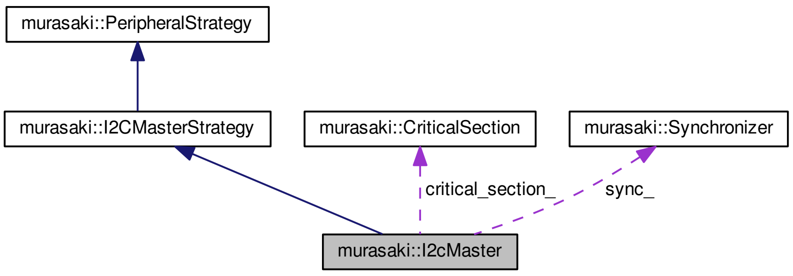
Thread safe, synchronous, and blocking IO. Encapsulating I2C master. Based on STM32Cube HAL driver and FreeRTOS.

```
#include <i2cmaster.hpp>
```

Inheritance diagram for murasaki::I2cMaster:



Collaboration diagram for murasaki::I2cMaster:


murasaki::PeripheralStrategy
murasaki::I2CMasterStrategy
murasaki::CriticalSection
murasaki::Synchronizer
critical_section_
sync_
murasaki::I2cMaster

Public Member Functions

- I2cMaster (I2C_HandleTypeDef ∗i2c_handle)

  Constructor.
- virtual ∼I2cMaster ()

  Destructor.
- virtual murasaki::I2cStatus Transmit (unsigned int addrs, const uint8_t ∗tx_data, unsigned int tx_size, unsigned int ∗transfered_count, unsigned int timeout_ms)

  Thread safe, synchronous transmission over I2C.
- virtual murasaki::I2cStatus Receive (unsigned int addrs, uint8_t ∗rx_data, unsigned int rx_size, unsigned int ∗transfered_count, unsigned int timeout_ms)

  Thread safe, synchronous receiving over I2C.
- virtual murasaki::I2cStatus TransmitThenReceive (unsigned int addrs, const uint8_t ∗tx_data, unsigned int tx_size, uint8_t ∗rx_data, unsigned int rx_size, unsigned int ∗tx_transfered_count, unsigned int ∗rx_←transfered_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.

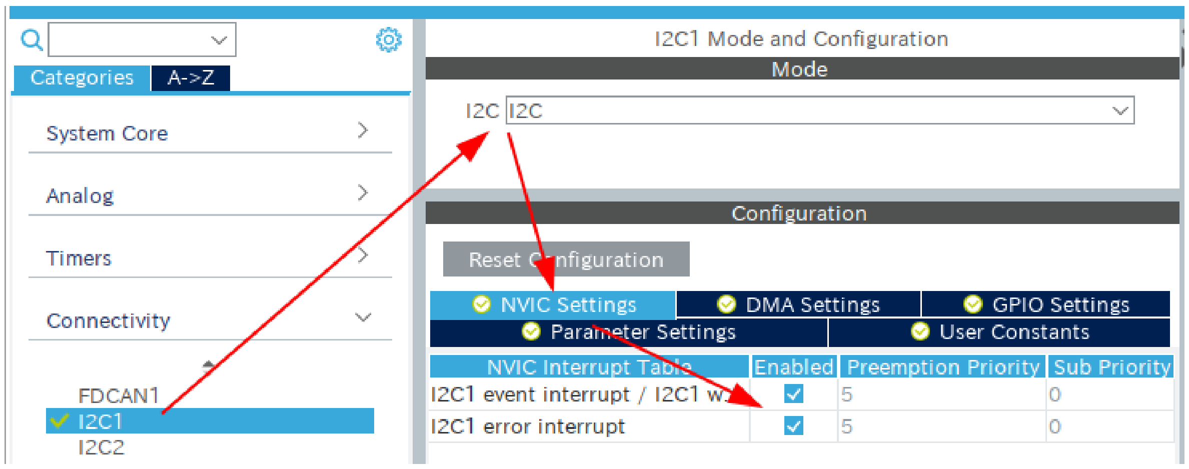
Additional Inherited Members

7.18.1 Detailed Description

The I2cMaster class is the wrapper of the I2C controller.

Configuration

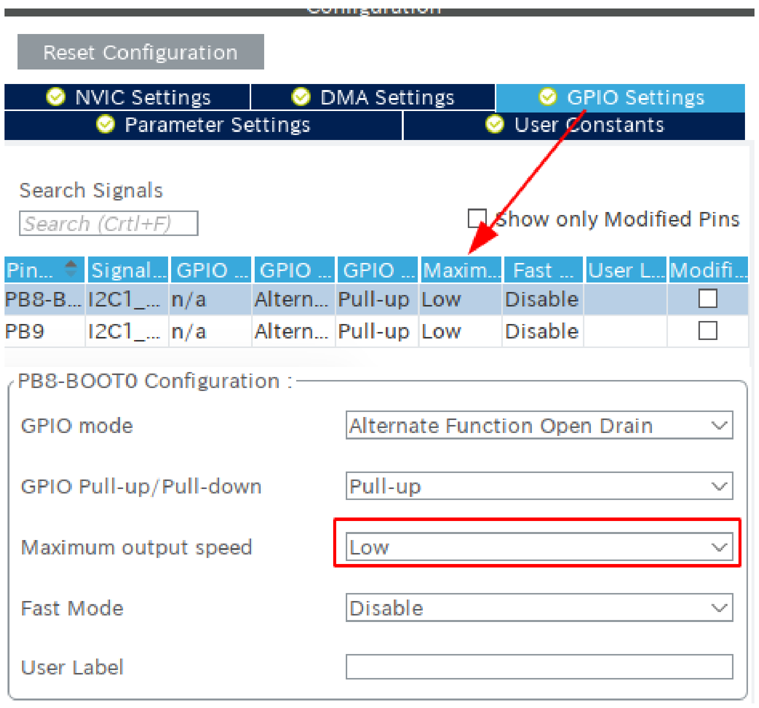
To configure the I2C peripheral as master, chose I2C peripheral in the Device Configuration Tool of the CubeIDE. Set it as I2C mode, and enable NVIC interrupt.



I2C: I2C

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
I2C1 event interrupt / I2C1 w...	✓	5	0
I2C1 error interrupt	✓	5	0

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.



Pin...	Signal...	GPIO ...	GPIO ...	GPIO ...	Maxim...	Fast ...	User L...	Modifi...
PB8-B...	I2C1_...	n/a	Altern...	Pull-up	Low	Disable		☐
PB9	I2C1_...	n/a	Altern...	Pull-up	Low	Disable		☐

PB8-BOOT0 Configuration :

GPIO mode	Alternate Function Open Drain
GPIO Pull-up/Pull-down	Pull-up
Maximum output speed	Low
Fast Mode	Disable
User Label	

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

In addition to the instantiation, we need to prepare an interrupt callback.

```
void HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef * hi2c)
{
    if (my_i2c3->TransmitCompleteCallback(hi2c))
        return;
}
```

Where HAL_I2C_SlaveTxCpltCallback() is a predefined name of the I2C interrupt handler. This function() is invoked by the system whenever an interrupt based I2C transmission is complete.

Note that the above callback is invoked by the system for any I2Cn where n is 1, 2, 3... To avoid the confusion, I2cMaster::TransmitCompleteCallback() method checks whether given parameter matches with its I2C_Handle↩ TypeDef ∗ pointer (which was passed to the constructor). And only when both matches, the member function execute the interrupt termination process and return with true.

As same as Tx, RX needs HAL_I2C_MasterRxCpltCallback() and Error needs HAL_I2C_ErrorCallback(). The H↩ AL_I2C_ErrorCallback() is essential to implement. Otherwise, I2C NAK response will not be handled correctly.

```
void HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c) {
    if (my_i2c3->HandleError(hi2c))
        return;
}
```

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.

7.18.2 Constructor & Destructor Documentation

7.18.2.1 murasaki::I2cMaster::I2cMaster (I2C_HandleTypeDef ∗ *i2c_handle*)

Parameters

i2c_handle	Peripheral handle created by CubeMx

7.18.3 Member Function Documentation

7.18.3.1 bool murasaki::I2cMaster::HandleError (void * *ptr*) [virtual]

Parameters

ptr	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::I2CMasterStrategy.

7.18.3.2 murasaki::I2cStatus murasaki::I2cMaster::Receive (unsigned int *addrs,* uint8_t * *rx_data,* unsigned int *rx_size,* unsigned int * *transfered_count,* unsigned int *timeout_ms*) [virtual]

Parameters

addrs	7bit address of the I2C device.
rx_data	Data array to transmit.
rx_size	Data counts[bytes] to transmit. Must be smaller than 65536
transfered_count	(Currently, Just ignored) the count of the bytes transfered during the API execution.
timeout_ms	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.

7.18.3.3 bool murasaki::I2cMaster::ReceiveCompleteCallback (void ∗ *ptr*) [virtual]

Parameters

ptr	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::I2CMasterStrategy.

7.18.3.4 murasaki::I2cStatus murasaki::I2cMaster::Transmit (unsigned int *addrs,* const uint8_t ∗ *tx_data,* unsigned int *tx_size,* unsigned int ∗ *transfered_count,* unsigned int *timeout_ms*) [virtual]

Parameters

addrs	7bit address of the I2C device.
tx_data	Data array to transmit.
tx_size	Data counts[bytes] to transmit. Must be smaller than 65536
transfered_count	(Currently, Just ignored) the count of the bytes transfered during the API execution.
timeout_ms	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.

7.18.3.5 bool murasaki::I2cMaster::TransmitCompleteCallback (void ∗ *ptr*) [virtual]

Parameters

ptr	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_MasterTxCpltCallback(),

```
void HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef * hi2c)
{
    if (murasaki::platform.i2c_master->TransmitCompleteCallback(hi2c))
        return;
}
```

Implements murasaki::I2CMasterStrategy.

7.18.3.6 murasaki::I2cStatus murasaki::I2cMaster::TransmitThenReceive (unsigned int *addrs,* const uint8_t ∗ *tx_data,* unsigned int *tx_size,* uint8_t ∗ *rx_data,* unsigned int *rx_size,* unsigned int ∗ *tx_transfered_count,* unsigned int ∗ *rx_transfered_count,* unsigned int *timeout_ms*) [virtual]

Parameters

addrs	7bit address of the I2C device.
tx_data	Data array to transmit.
tx_size	Data counts[bytes] to transmit. Must be smaller than 65536
rx_data	Data array to transmit.
rx_size	Data counts[bytes] to transmit. Must be smaller than 65536
tx_transfered_count	(Currently, Just ignored) the count of the bytes transmitted during the API execution.
rx_transfered_count	(Currently, Just ignored) the count of the bytes received during the API execution.
timeout_ms	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:

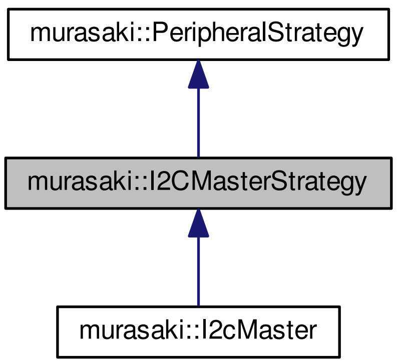
- i2cmaster.hpp
- i2cmaster.cpp

7.19 murasaki::I2CMasterStrategy Class Reference

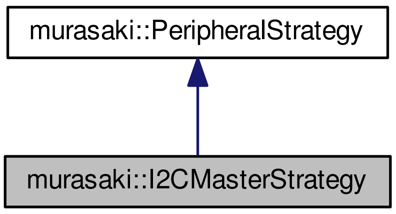
Definition of the root class of I2C master.

```
#include <i2cmasterstrategy.hpp>
```

Inheritance diagram for murasaki::I2CMasterStrategy:



Collaboration diagram for murasaki::I2CMasterStrategy:



Public Member Functions

- virtual murasaki::I2cStatus Transmit (unsigned int addrs, const uint8_t ∗tx_data, unsigned int tx_size, unsigned int ∗transfered_count=nullptr, unsigned int timeout_ms=murasaki::kwmsIndefinitely)=0

    Thread safe, synchronous transmission over I2C.
- virtual murasaki::I2cStatus Receive (unsigned int addrs, uint8_t ∗rx_data, unsigned int rx_size, unsigned int ∗transfered_count=nullptr, unsigned int timeout_ms=murasaki::kwmsIndefinitely)=0

    Thread safe, synchronous receiving over I2C.
- virtual murasaki::I2cStatus TransmitThenReceive (unsigned int addrs, const uint8_t ∗tx_data, unsigned int tx_size, uint8_t ∗rx_data, unsigned int rx_size, unsigned int ∗tx_transfered_count=nullptr, unsigned int ∗rx←_transfered_count=nullptr, unsigned int timeout_ms=murasaki::kwmsIndefinitely)=0

    Thread safe, synchronous transmission and then receiving over I2C.
- 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

7.19.1 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.

7.19.2 Member Function Documentation

7.19.2.1 virtual bool murasaki::I2CMasterStrategy::HandleError (void ∗ *ptr*) [pure virtual]

Parameters

ptr	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.

7.19.2.2 virtual **murasaki::I2cStatus** murasaki::I2CMasterStrategy::Receive (unsigned int *addrs,* uint8_t ∗ *rx_data,* unsigned int *rx_size,* unsigned int ∗ *transfered_count* = nullptr*,* unsigned int *timeout_ms* = **murasaki::kwmsIndefinitely**) [pure virtual]

Parameters

addrs	7bit address of the I2C device.
rx_data	Data array to transmit.
rx_size	Data counts[bytes] to transmit.
transfered_count	the count of the bytes transfered during the API execution.
timeout_ms	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.

7.19.2.3 virtual bool murasaki::I2CMasterStrategy::ReceiveCompleteCallback (void ∗ *ptr*) [pure virtual]

Parameters

ptr	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.

7.19.2.4 virtual **murasaki::I2cStatus** murasaki::I2CMasterStrategy::Transmit (unsigned int *addrs*, const uint8_t ∗ *tx_data*, unsigned int *tx_size*, unsigned int ∗ *transfered_count* = `nullptr`, unsigned int *timeout_ms* = **murasaki::kwmsIndefinitely**) `[pure virtual]`

Parameters

addrs	7bit address of the I2C device.
tx_data	Data array to transmit.
tx_size	Data counts[bytes] to transmit.
transfered_count	the count of the bytes transfered during the API execution.
timeout_ms	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.

7.19.2.5 virtual bool murasaki::I2CMasterStrategy::TransmitCompleteCallback (void ∗ *ptr*) `[pure virtual]`

Parameters

ptr	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.

7.19.2.6 virtual **murasaki::I2cStatus** murasaki::I2CMasterStrategy::TransmitThenReceive (unsigned int *addrs*, const uint8_t ∗ *tx_data*, unsigned int *tx_size*, uint8_t ∗ *rx_data*, unsigned int *rx_size*, unsigned int ∗ *tx_transfered_count* = `nullptr`, unsigned int ∗ *rx_transfered_count* = `nullptr`, unsigned int *timeout_ms* = **murasaki::kwmsIndefinitely**) `[pure virtual]`

Parameters

addrs	7bit address of the I2C device.
tx_data	Data array to transmit.
tx_size	Data counts[bytes] to transmit.
rx_data	Data array to transmit.
rx_size	Data counts[bytes] to transmit.
tx_transfered_count	the count of the bytes transmitted during the API execution.
rx_transfered_count	the count of the bytes received during the API execution.
timeout_ms	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:

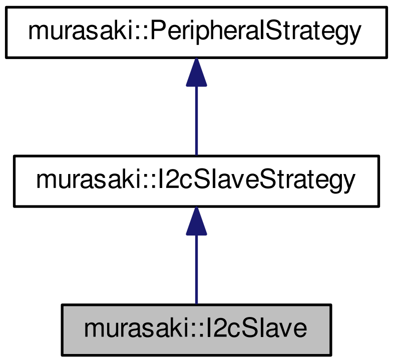
- i2cmasterstrategy.hpp

7.20 murasaki::I2cSlave Class Reference

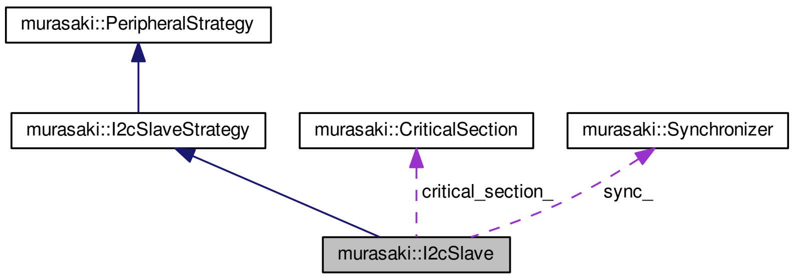
Thread safe, synchronous and blocking IO. Encapsulating I2C slave. Based on STM32Cube HAL driver and Free↩ RTOS.

`#include <i2cslave.hpp>`

Inheritance diagram for murasaki::I2cSlave:



Collaboration diagram for murasaki::I2cSlave:


murasaki::PeripheralStrategy
murasaki::I2cSlaveStrategy
murasaki::CriticalSection
murasaki::Synchronizer
critical_section_
sync_
murasaki::I2cSlave

Public Member Functions

- virtual murasaki::I2cStatus Transmit (const uint8_t ∗tx_data, unsigned int tx_size, unsigned int ∗transfered↩
_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 ∗transfered_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

7.20.1 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

In addition to the instantiation, we need to prepare an interrupt callback. and error callback

```
void HAL_I2C_SlaveTxCpltCallback(I2C_HandleTypeDef * hi2c)
{
   if ( my_i2c3->TransmitCompleteCallback(hi2c))
     return;
}
```

Where HAL_I2C_SlaveTxCpltCallback() is a predefined name of the I2C interrupt handler. This function is invoked by the system whenever an interrupt based I2C transmission is complete. Because the default function is weakly bound, the above definition overrides the default one.

Note that Any I2Cn where n is 1, 2, 3, ... call the above callback function. To avoid the confusion, I2cMaster↩
::TransmitCompleteCallback() method checks whether given parameter matches with its I2C_HandleTypeDef ∗ pointer (which was passed to the constructor). And only when both matches, the member function execute the interrupt termination process. In the case of the successful match, it returns true.

As same as Tx, RX needs HAL_I2C_SlaveRxCpltCallback() and Error needs HAL_I2C_ErrorCallback(). The HA↩
L_I2C_ErrorCallback() is essential to implement. Otherwise, NAK response will not be handled correctly.

```
void HAL_I2C_ErrorCallback(I2C_HandleTypeDef * hi2c)
{
   if (my_i2c3->HandleError(hi2c))
      return;

}
```

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_I2↩C_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.

7.20.2 Member Function Documentation

7.20.2.1 bool murasaki::I2cSlave::HandleError (void ∗ *ptr*) [virtual]

Parameters

ptr	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::I2cSlaveStrategy.

7.20.2.2 murasaki::I2cStatus murasaki::I2cSlave::Receive (uint8_t ∗ *rx_data,* unsigned int *rx_size,* unsigned int ∗ *transfered_count,* unsigned int *timeout_ms*) [virtual]

Parameters

rx_data	Data array to transmit.
rx_size	Data counts[bytes] to transmit. Must be smaller than 65536
transfered_count	(Currently, Just ignored) the count of the bytes transferred during the API execution.
timeout_ms	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.

7.20.2.3 bool murasaki::I2cSlave::ReceiveCompleteCallback (void ∗ *ptr*) [virtual]

Parameters

ptr	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::I2cSlaveStrategy.

7.20.2.4 murasaki::I2cStatus murasaki::I2cSlave::Transmit (const uint8_t ∗ *tx_data,* unsigned int *tx_size,* unsigned int ∗ *transfered_count,* unsigned int *timeout_ms*) [virtual]

Parameters

tx_data	Data array to transmit.
tx_size	Data counts[bytes] to transmit. Must be smaller than 65536
transfered_count	(Currently, Just ignored) the count of the bytes transferred during the API execution.
timeout_ms	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.

7.20.2.5 bool murasaki::I2cSlave::TransmitCompleteCallback (void ∗ *ptr*) [virtual]

Parameters

ptr	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::I2cSlaveStrategy.

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

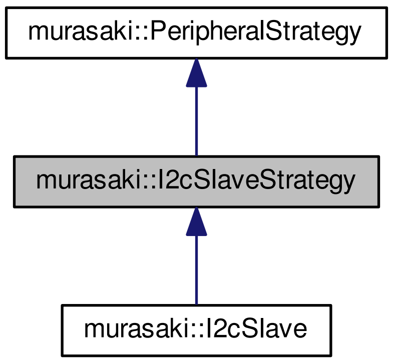
- i2cslave.hpp
- i2cslave.cpp

7.21 murasaki::I2cSlaveStrategy Class Reference

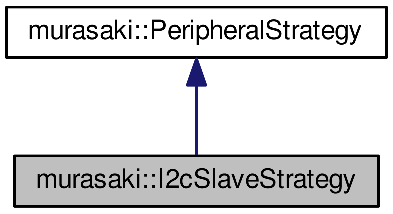
Definition of the root class of I2C Slave.

`#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 ∗transfered←_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 ∗transfered_←count=nullptr, unsigned int timeout_ms=murasaki::kwmsIndefinitely)=0

  Thread safe, synchronous receiving over I2C.
- 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

7.21.1 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.

7.21.2 Member Function Documentation

7.21.2.1 virtual bool murasaki::I2cSlaveStrategy::HandleError (void ∗ *ptr*) `[pure virtual]`

Parameters

ptr	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 is a member function to detect error.

The error handling is depend on the implementation.

Implemented in murasaki::I2cSlave.

7.21.2.2 virtual **murasaki::I2cStatus** murasaki::I2cSlaveStrategy::Receive (uint8_t ∗ *rx_data*, unsigned int *rx_size*, unsigned int ∗ *transfered_count* = `nullptr`, unsigned int *timeout_ms* = **murasaki::kwmsIndefinitely**) `[pure virtual]`

Parameters

rx_data	Data array to transmit.
rx_size	Data counts[bytes] to transmit.
transfered_count	the count of the bytes transferred during the API execution.
timeout_ms	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.

7.21.2.3 virtual bool murasaki::I2cSlaveStrategy::ReceiveCompleteCallback (void ∗ *ptr*) `[pure virtual]`

Parameters

ptr	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 call back 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::I2cSlave.

7.21.2.4 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

tx_data	Data array to transmit.
tx_size	Data counts[bytes] to transmit.
transfered_count	the count of the bytes transferred during the API execution.
timeout_ms	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.

7.21.2.5 virtual bool murasaki::I2cSlaveStrategy::TransmitCompleteCallback (void * *ptr*) `[pure virtual]`

Parameters

ptr	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 call back 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::I2cSlave.

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

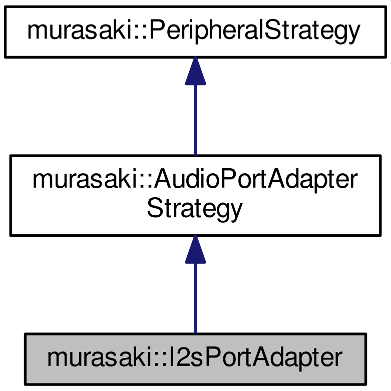
- i2cslavestrategy.hpp

7.22 murasaki::I2sPortAdapter Class Reference

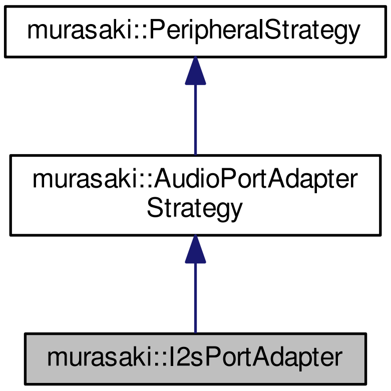
Adapter as I2S audio port.

```
#include <i2sportadapter.hpp>
```

Inheritance diagram for murasaki::I2sPortAdapter:


murasaki::PeripheralStrategy
murasaki::AudioPortAdapter Strategy
murasaki::I2sPortAdapter

Collaboration diagram for murasaki::I2sPortAdapter:


murasaki::PeripheralStrategy
murasaki::AudioPortAdapter Strategy
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

7.22.1 Detailed Description

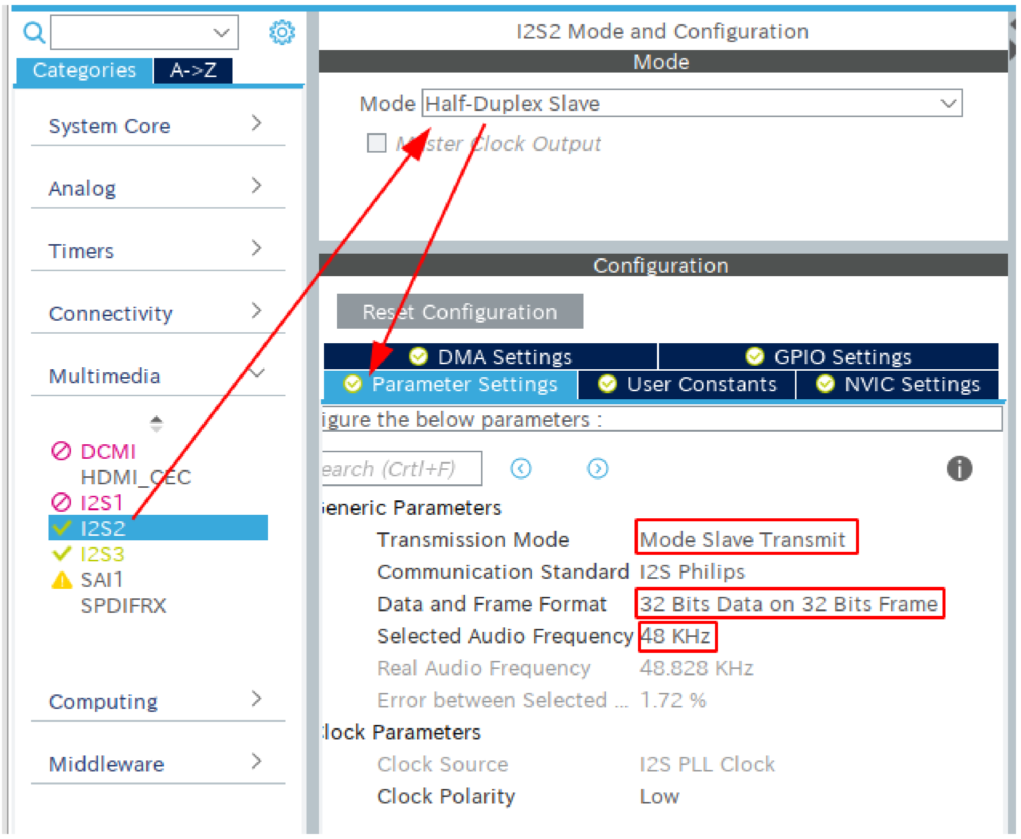
This class is a dedicated adapter class for the murasaki::DuplexAudio. The DuplexAudio class can handle audio through the I2S port by using this adapter.

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.



I2S2 Mode and Configuration

Mode

Mode Half-Duplex Slave

Master Clock Output

Configuration

igure the below parameters :

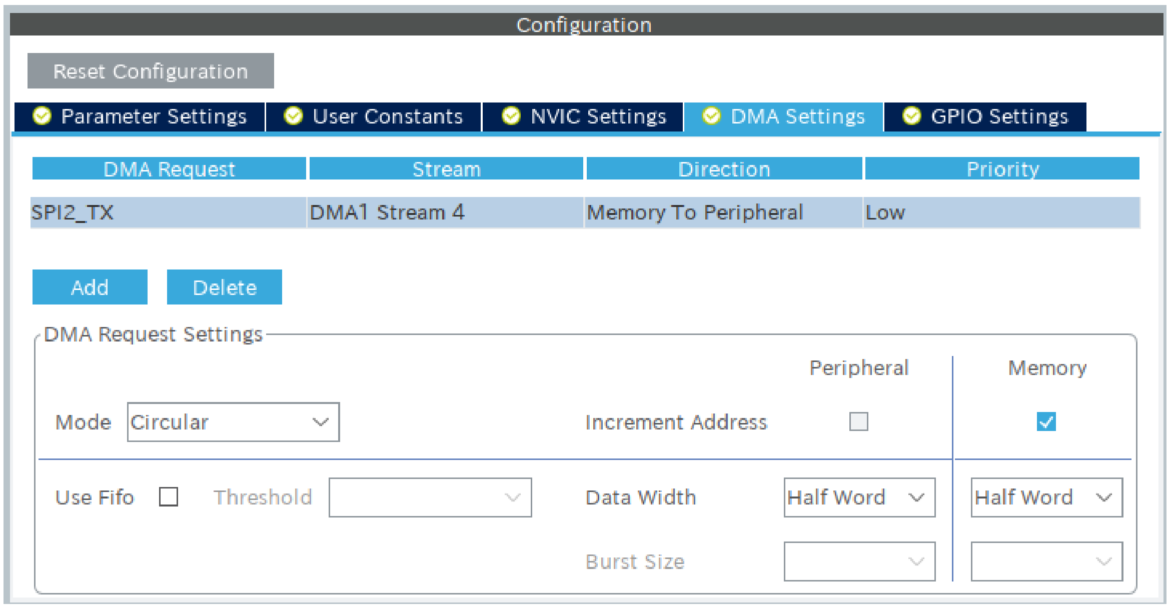
ieneric Parameters

Transmission Mode	Mode Slave Transmit
Communication Standard	I2S Philips
Data and Frame Format	32 Bits Data on 32 Bits Frame
Selected Audio Frequency	48 KHz
Real Audio Frequency	48.828 KHz
Error between Selected ...	1.72 %

lock Parameters

Clock Source	I2S PLL Clock
Clock Polarity	Low

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.



DMA Request	Stream	Direction	Priority
SPI2_TX	DMA1 Stream 4	Memory To Peripheral	Low

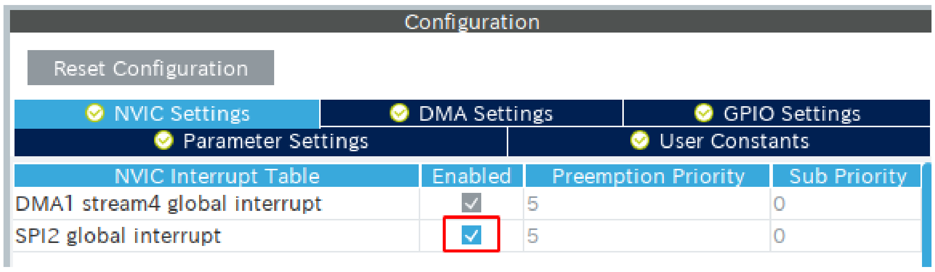
DMA Request Settings

	Peripheral	Memory
Increment Address	☐	☑
Data Width	Half Word	Half Word
Burst Size		

Mode Circular

Use Fifo ☐ Threshold

And then enable the interrupt if the peripheral is a receiver. Enabling the interrupt for a transmitter does not affect.


Configuration

Reset Configuration

NVIC Settings | DMA Settings | GPIO Settings
Parameter Settings | User Constants

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
DMA1 stream4 global interrupt	✓	5	0
SPI2 global interrupt	✓	5	0

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 );
```

7.22.2 Constructor & Destructor Documentation

7.22.2.1 murasaki::I2sPortAdapter::I2sPortAdapter (I2S_HandleTypeDef ∗ *tx_peripheral,* I2S_HandleTypeDef ∗ *rx_peripheral*)

Parameters

tx_peripheral	I2S_HandleTypeDef type peripheral for TX. This is defined in main.c.
rx_peripheral	I2S_HandleTypeDef type peripheral for RX. This is defined in main.c.

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.

7.22.3 Member Function Documentation

7.22.3.1 virtual unsigned int murasaki::I2sPortAdapter::GetNumberOfChannelsRx ( ) [inline], [virtual]

Returns

always 2

Implements murasaki::AudioPortAdapterStrategy.

7.22.3.2 virtual unsigned int murasaki::I2sPortAdapter::GetNumberOfChannelsTx ( ) [inline], [virtual]

Returns

always 2

Implements murasaki::AudioPortAdapterStrategy.

7.22.3.3 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.

7.22.3.4 void ∗ murasaki::I2sPortAdapter::GetPeripheralHandle ( ) [virtual]

Returns

pointer to the raw peripheral handler hidden in a class.

Implements murasaki::AudioPortAdapterStrategy.

7.22.3.5 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 formant 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.

7.22.3.6 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 formant 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.

7.22.3.7 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.

7.22.3.8 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.

7.22.3.9 bool murasaki::I2sPortAdapter::HandleError (void ∗ *ptr*) [virtual]

Parameters

ptr	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.

7.22.3.10 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.

7.22.3.11 bool murasaki::I2sPortAdapter::Match (void ∗ *peripheral_handle*) [virtual]

Check if peripheral handle matched with given handle.

Parameters

peripheral_handle	

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.

7.22.3.12 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.

7.22.3.13 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:

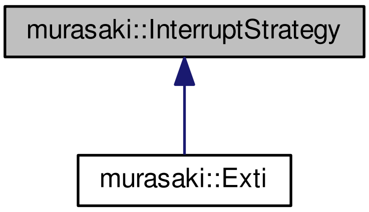
- i2sportadapter.hpp
- i2sportadapter.cpp

7.23 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

7.23.1 Detailed Description

This is a mother of the interrupt classes. The interrut class provides following functionalities :

- Enable/ Disable interrupt line by line.
- Synchronization between interrupt and task.

7.23.2 Member Function Documentation

7.23.2.1 virtual bool murasaki::InterruptStrategy::Release (unsigned int *line*) `[pure virtual]`

Release the waiting task

Parameters

line	The identifier of the interrupt

Returns

true if line is matched with this EXTI. false if not matched.

Implemented in murasaki::Exti.

7.23.2.2 virtual **murasaki::InterruptStatus** murasaki::InterruptStrategy::Wait (unsigned int *timeout* = **murasaki::kwmsIndefinitely**) `[pure virtual]`

Wait for an interrupt from task.

Parameters

timeout	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:

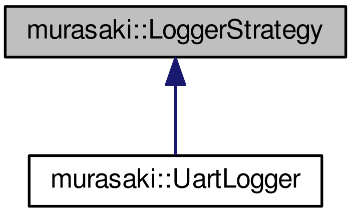
- interruptstrategy.hpp

7.24 murasaki::LoggerStrategy Class Reference

Abstract class for logging.

```
#include <loggerstrategy.hpp>
```

Inheritance diagram for murasaki::LoggerStrategy:



Public Member Functions

- virtual ∼LoggerStrategy ()

  Detructor.
- 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.

7.24.1 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.

7.24.2 Constructor & Destructor Documentation

7.24.2.1 virtual murasaki::LoggerStrategy::∼LoggerStrategy ( ) [inline], [virtual]

Do nothing here. Declared to enforce the derived class's constructor as "virtual".

7.24.3 Member Function Documentation

7.24.3.1 virtual void murasaki::LoggerStrategy::DoPostMortem (void ∗ *debugger_fifo*) [inline], [virtual]

Parameters

debugger_fifo	Pointer to the DebuggerFifo 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::SetPost←Mortem() internally.

Reimplemented in murasaki::UartLogger.

7.24.3.2 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.

7.24.3.3 virtual void murasaki::LoggerStrategy::putMessage (char *message[ ]*, unsigned int *size*) [pure virtual]

Parameters

message	Non null terminated character array. This data is stored or output to the logger.
size	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:

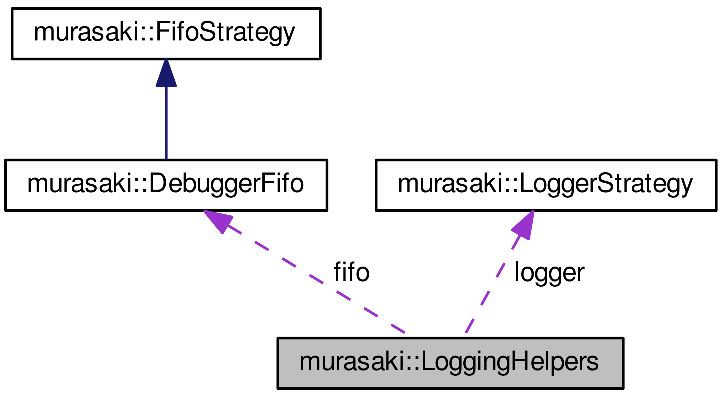
- loggerstrategy.hpp

7.25 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:

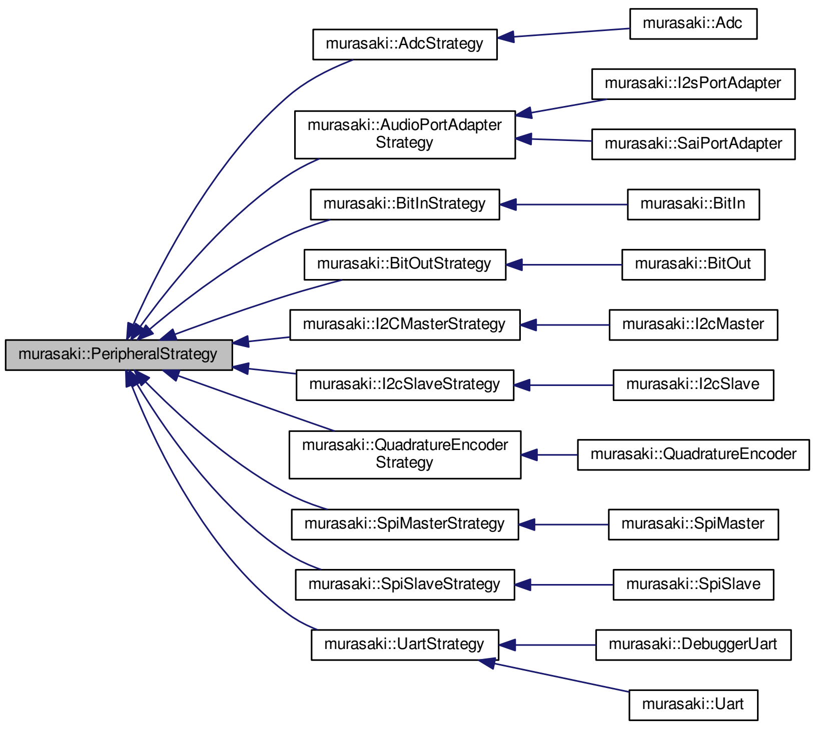
- debuggerfifo.hpp

7.26 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

7.26.1 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.

7.26.2 Member Function Documentation

7.26.2.1 virtual void∗ murasaki::PeripheralStrategy::GetPeripheralHandle ( ) [protected], [pure virtual]

Returns

pointer to the raw peripheral handler hidden in a class.

Implemented in murasaki::I2sPortAdapter, murasaki::SaiPortAdapter, murasaki::Adc, murasaki::AudioPort←AdapterStrategy, murasaki::BitOut, and murasaki::BitIn.

7.26.2.2 virtual bool murasaki::PeripheralStrategy::Match (void ∗ *peripheral_handle*) [inline], [virtual]

Check if peripheral handle matched with given handle.

Parameters

peripheral_handle	

Returns

true if match, false if not match.

Reimplemented in murasaki::I2sPortAdapter, murasaki::SaiPortAdapter, and murasaki::AudioPortAdapterStrategy.

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

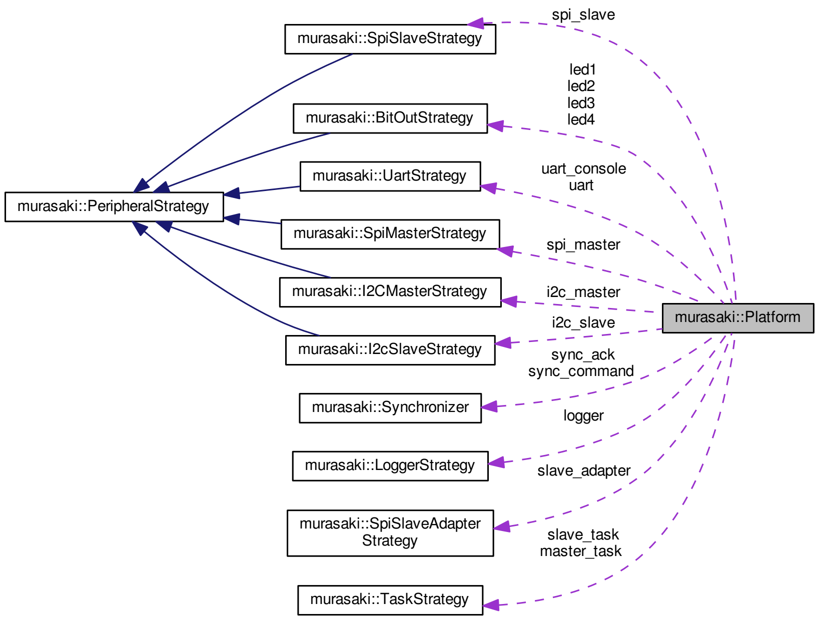
- peripheralstrategy.hpp

7.27 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.

7.27.1 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:

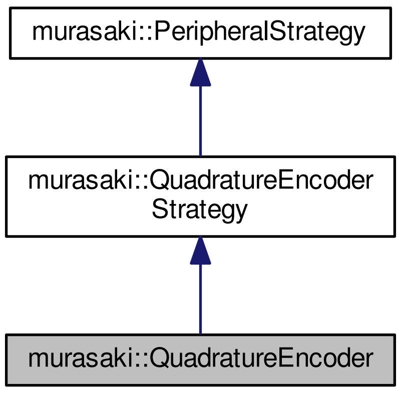
- platform_defs.hpp

7.28 murasaki::QuadratureEncoder Class Reference

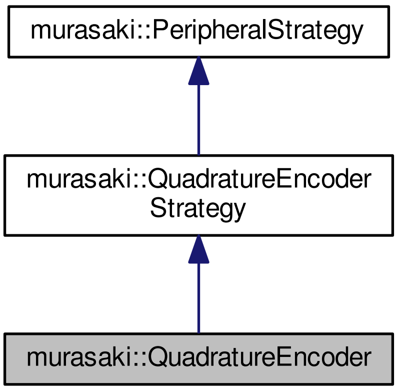
Quadrature Encoder class.

```
#include <quadratureencoder.hpp>
```

Inheritance diagram for murasaki::QuadratureEncoder:


murasaki::PeripheralStrategy
murasaki::QuadratureEncoder Strategy
murasaki::QuadratureEncoder

Collaboration diagram for murasaki::QuadratureEncoder:


murasaki::PeripheralStrategy
murasaki::QuadratureEncoder Strategy
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

7.28.1 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 encoer allow to up / down the counter by CW/CCW rotation. Basic parameter have to be set by CubeIDE/CubeMX

7.28.2 Constructor & Destructor Documentation

7.28.2.1 murasaki::QuadratureEncoder::QuadratureEncoder ( ) [delete]

Suppress the default constructor.

7.28.2.2 murasaki::QuadratureEncoder::QuadratureEncoder (TIM_HandleTypeDef ∗ *htim*)

Parameters

htim	The pointer to the peripheral control structure.

The given parameter is stored to the peripheral_ variable.

7.28.3 Member Function Documentation

7.28.3.1 unsigned int murasaki::QuadratureEncoder::Get (void) [virtual]

Returns

The value of the internal counter.

Return the encoder internal value.

Implements murasaki::QuadratureEncoderStrategy.

7.28.3.2 void murasaki::QuadratureEncoder::Set (unsigned int *value*) [virtual]

Parameters

value	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:

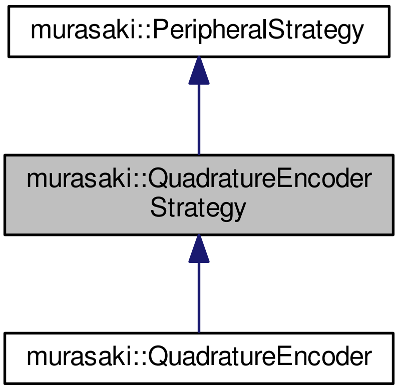
- quadratureencoder.hpp
- quadratureencoder.cpp

7.29 murasaki::QuadratureEncoderStrategy Class Reference

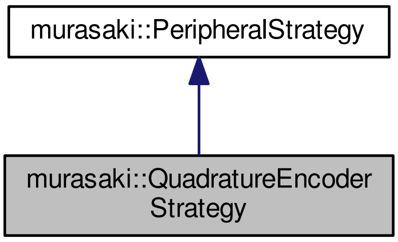
Strategy class for the quadrature encoder.

```
#include <quadratureencoderstrategy.hpp>
```

Inheritance diagram for murasaki::QuadratureEncoderStrategy:


murasaki::PeripheralStrategy
murasaki::QuadratureEncoder Strategy
murasaki::QuadratureEncoder

Collaboration diagram for murasaki::QuadratureEncoderStrategy:


murasaki::PeripheralStrategy
murasaki::QuadratureEncoder Strategy

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

7.29.1 Detailed Description

The strategy class of the quadrature encoder peripheral. The inherited class will control the encoder.

7.29.2 Member Function Documentation

7.29.2.1 virtual unsigned int murasaki::QuadratureEncoderStrategy::Get ( ) `[pure virtual]`

Returns

The value of the internal counter.

Return the encoder internal value.

Implemented in murasaki::QuadratureEncoder.

7.29.2.2 virtual void murasaki::QuadratureEncoderStrategy::Set (unsigned int *value*) `[pure virtual]`

Parameters

value	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:

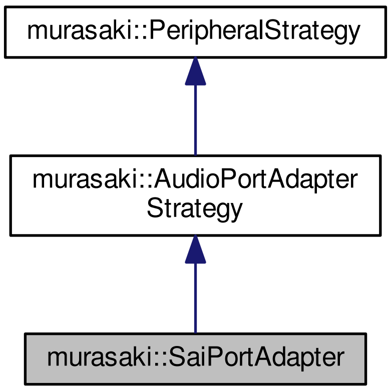
- quadratureencoderstrategy.hpp

7.30 murasaki::SaiPortAdapter Class Reference

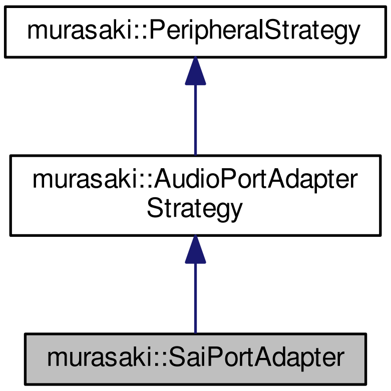
Adapter as SAI audio port.

`#include <saiportadapter.hpp>`

Inheritance diagram for murasaki::SaiPortAdapter:


murasaki::PeripheralStrategy
murasaki::AudioPortAdapter Strategy
murasaki::SaiPortAdapter

Collaboration diagram for murasaki::SaiPortAdapter:


murasaki::PeripheralStrategy
murasaki::AudioPortAdapter Strategy
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

7.30.1 Detailed Description

Dedicated adapter for the murasaki::DuplexAudio. By passing this adapter, the DuplexAudio class can handle audio through the SAI port.

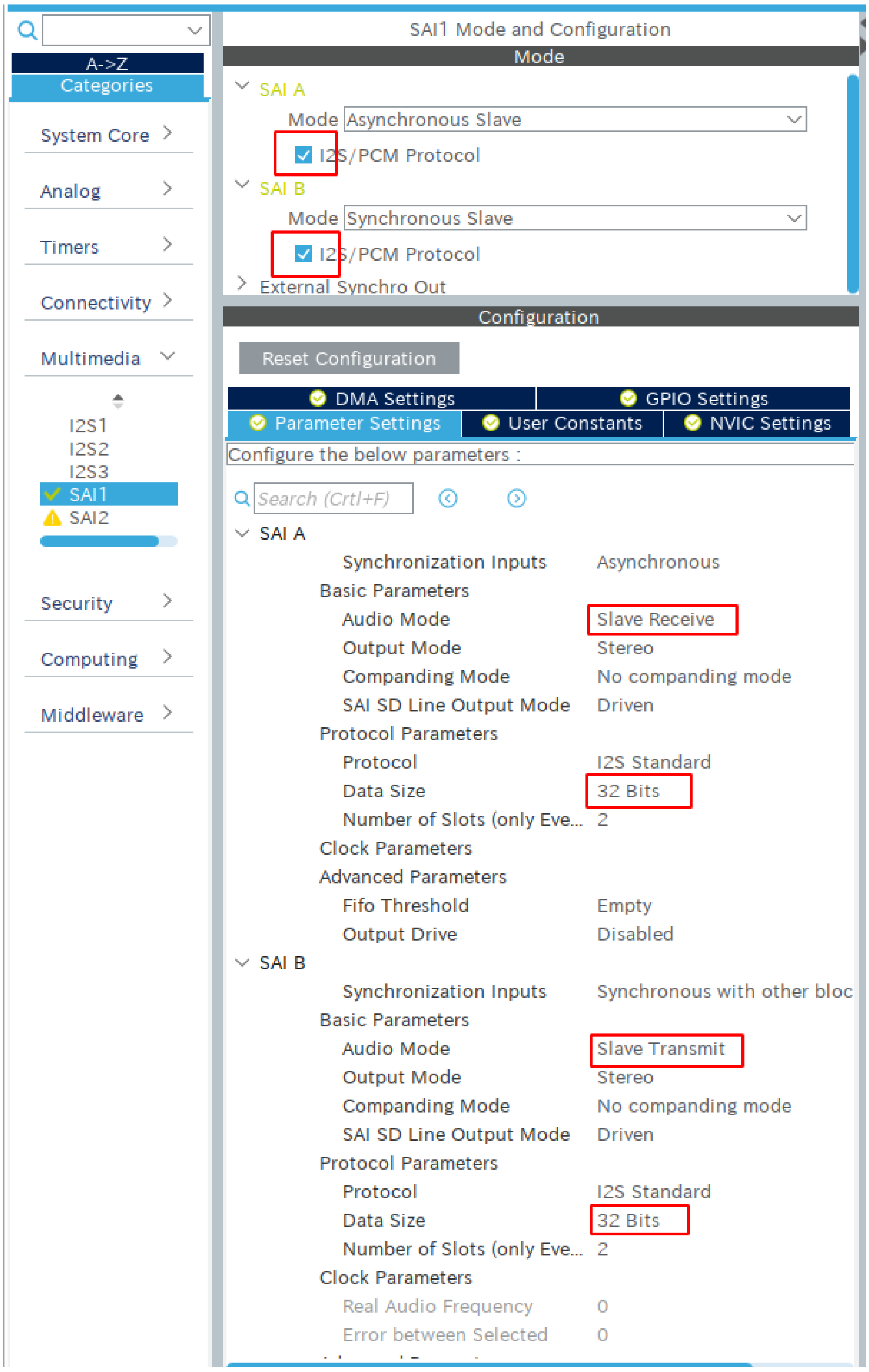
Caution : The size of the data in SAI and the width of the data in DMA must be aligned. This is responsibility of the programmer. The mis-aligned configuration gives broken audio.

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.



SAI1 Mode and Configuration

Mode

SAI A

Mode Asynchronous Slave

✓ I2S/PCM Protocol

SAI B

Mode Synchronous Slave

✓ I2S/PCM Protocol

External Synchro Out

Configuration

Configure the below parameters :

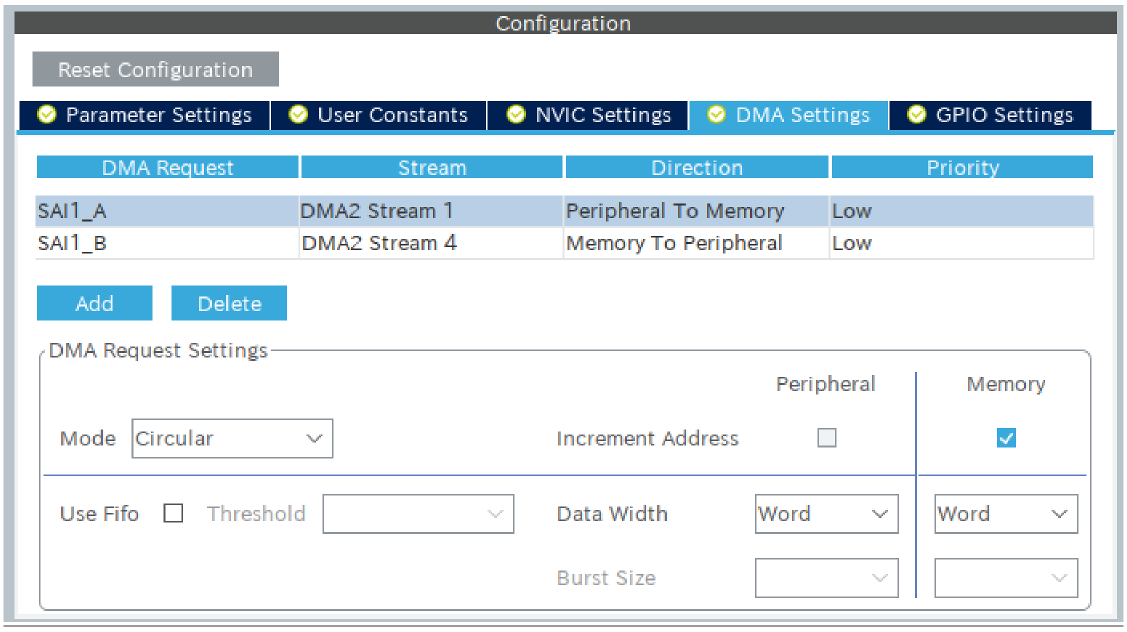
SAI A

Synchronization Inputs	Asynchronous
Basic Parameters	
Audio Mode	Slave Receive
Output Mode	Stereo
Companding Mode	No companding mode
SAI SD Line Output Mode	Driven
Protocol Parameters	
Protocol	I2S Standard
Data Size	32 Bits
Number of Slots (only Eve...	2
Clock Parameters	
Advanced Parameters	
Fifo Threshold	Empty
Output Drive	Disabled

SAI B

Synchronization Inputs	Synchronous with other bloc
Basic Parameters	
Audio Mode	Slave Transmit
Output Mode	Stereo
Companding Mode	No companding mode
SAI SD Line Output Mode	Driven
Protocol Parameters	
Protocol	I2S Standard
Data Size	32 Bits
Number of Slots (only Eve...	2
Clock Parameters	
Real Audio Frequency	0
Error between Selected	0

The configuration of DMA is tricky. Set the DMA based on the table below.



DMA Request	Stream	Direction	Priority
SAI1_A	DMA2 Stream 1	Peripheral To Memory	Low
SAI1_B	DMA2 Stream 4	Memory To Peripheral	Low

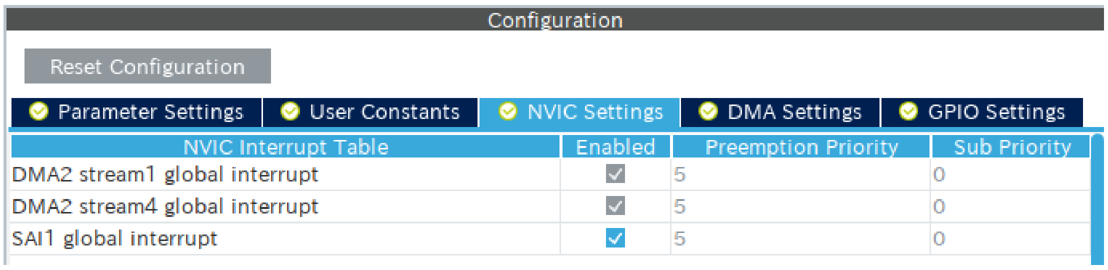
DMA Request Settings

	Peripheral	Memory
Increment Address	☐	✓
Data Width	Word	Word
Burst Size		

Mode: Circular

Use Fifo ☐ Threshold

And then, enable the interrupt .



NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
DMA2 stream1 global interrupt	✓	5	0
DMA2 stream4 global interrupt	✓	5	0
SAI1 global interrupt	✓	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 );
```

7.30.2 Constructor & Destructor Documentation

7.30.2.1 murasaki::SaiPortAdapter::SaiPortAdapter (SAI_HandleTypeDef ∗ *tx_peripheral,* SAI_HandleTypeDef ∗ *rx_peripheral*)

Parameters

tx_peripheral	SAI_HandleTypeDef type peripheral for TX. This is defined in main.c.
rx_peripheral	SAI_HandleTypeDef type peripheral for RX. This is defined in main.c.

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.

7.30.3 Member Function Documentation

7.30.3.1 unsigned int murasaki::SaiPortAdapter::GetNumberOfChannelsRx ( ) `[virtual]`

Returns

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

Implements murasaki::AudioPortAdapterStrategy.

7.30.3.2 unsigned int murasaki::SaiPortAdapter::GetNumberOfChannelsTx ( ) `[virtual]`

Returns

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

Implements murasaki::AudioPortAdapterStrategy.

7.30.3.3 virtual unsigned int murasaki::SaiPortAdapter::GetNumberOfDMAPhase ( ) `[inline]`, `[virtual]`

Returns

Always return 2 for STM32 SAI, becuase the cyclic DMA has halfway and complete interrupt.

Implements murasaki::AudioPortAdapterStrategy.

7.30.3.4 void ∗ murasaki::SaiPortAdapter::GetPeripheralHandle ( ) `[virtual]`

Returns

pointer to the raw peripheral handler hidden in a class.

Implements murasaki::AudioPortAdapterStrategy.

7.30.3.5 unsigned int murasaki::SaiPortAdapter::GetSampleShiftSizeRx ( ) [virtual]

Returns

0 The unit is [bit]

This is needed because of the mismatch in the DMA buffer data formant 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 shuld return 0.

The STM32 I2S DMA format is left aligned. So, always return 0.

Implements murasaki::AudioPortAdapterStrategy.

7.30.3.6 unsigned int murasaki::SaiPortAdapter::GetSampleShiftSizeTx ( ) [virtual]

Returns

Always 0.

This is needed because of the mismatch in the DMA buffer data formant 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 shuld return 0.

The STM32 SAI DMA format is right aligned.

Implements murasaki::AudioPortAdapterStrategy.

7.30.3.7 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.

7.30.3.8 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.

7.30.3.9 bool murasaki::SaiPortAdapter::HandleError (void * *ptr*) [virtual]

Parameters

ptr	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.

7.30.3.10 bool murasaki::SaiPortAdapter::IsInt16SwapRequired ( ) [virtual]

Returns

Always false.

Display whether the half word (int16_t) swap is required or not.

SAI doens't require the half word swap inside word. Thus, always returns false

Implements murasaki::AudioPortAdapterStrategy.

7.30.3.11 bool murasaki::SaiPortAdapter::Match (void ∗ *peripheral_handle*) [virtual]

Check if peripheral handle matched with given handle.

Parameters

peripheral_handle	

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.

7.30.3.12 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.

7.30.3.13 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:

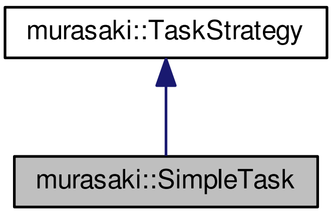
- saiportadapter.hpp
- saiportadapter.cpp

7.31 murasaki::SimpleTask Class Reference

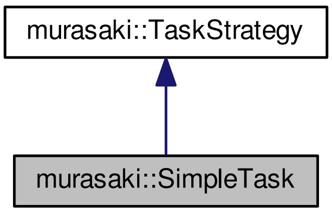
An easy to use task class.

`#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

7.31.1 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();
```

7.31.2 Constructor & Destructor Documentation

7.31.2.1 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

task_name	A name of task. This is relevant to the FreeRTOS's API manner.
stack_depth	Task stack size by byte.
task_priority	The task priority. Max priority is defined by configMAX_PRIOIRTIES in FreeRTOSConfig.h
task_parameter	A pointer to the parameter passed to task.
task_body_func	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.

7.31.3 Member Function Documentation

7.31.3.1 void murasaki::SimpleTask::TaskBody (const void * *ptr*) [protected], [virtual]

Parameters

ptr	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:

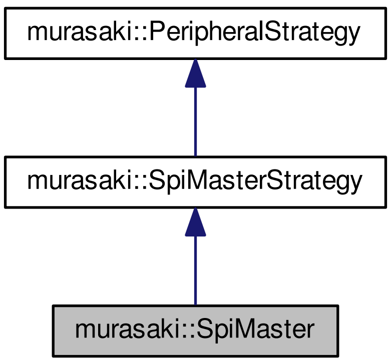
- simpletask.hpp
- simpletask.cpp

7.32 murasaki::SpiMaster Class Reference

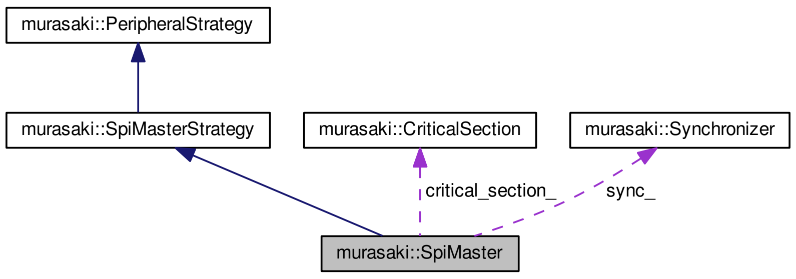
Thread safe, synchronous and blocking IO. Encapsulating SPI master. Based on STM32Cube HAL driver and FreeRTOS.

```
#include <spimaster.hpp>
```

Inheritance diagram for murasaki::SpiMaster:



Collaboration diagram for murasaki::SpiMaster:


murasaki::PeripheralStrategy
murasaki::SpiMasterStrategy
murasaki::CriticalSection
murasaki::Synchronizer
critical_section_
sync_
murasaki::SpiMaster

Public Member Functions

- SpiMaster (SPI_HandleTypeDef ∗spi_handle)

    Constractor.
- 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.

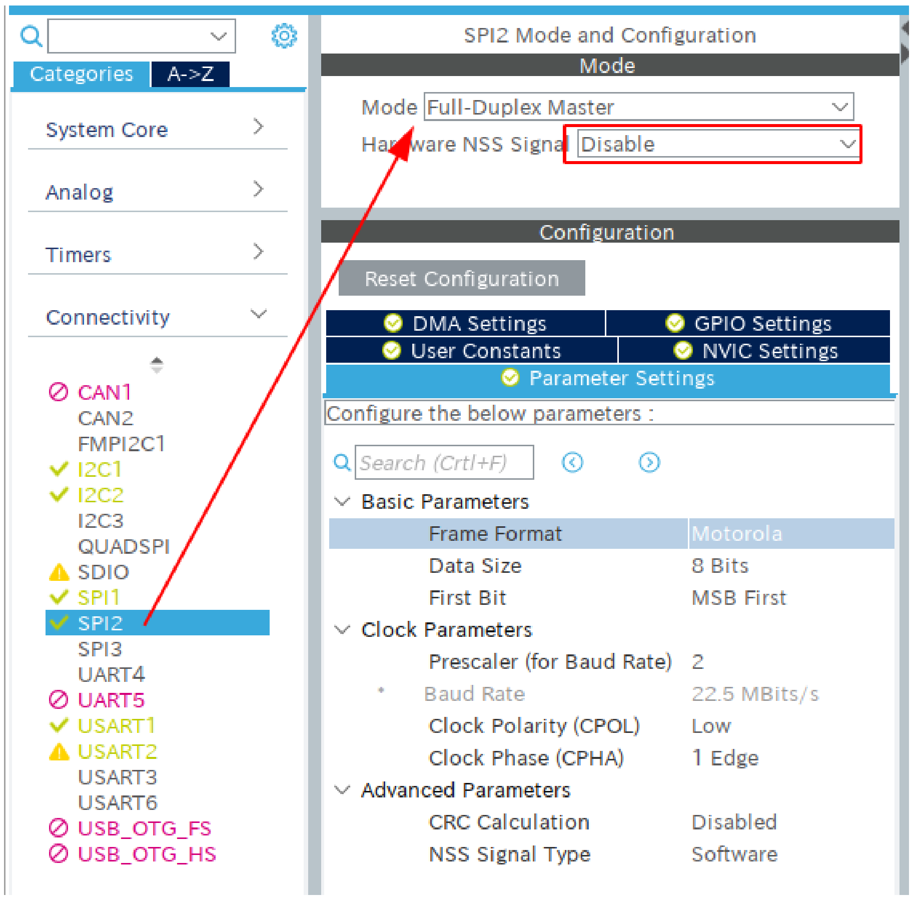
Additional Inherited Members

7.32.1 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



SPI2 Mode and Configuration

Mode

Mode: Full-Duplex Master

Hardware NSS Signal: Disable

Configuration

Categories | A->Z

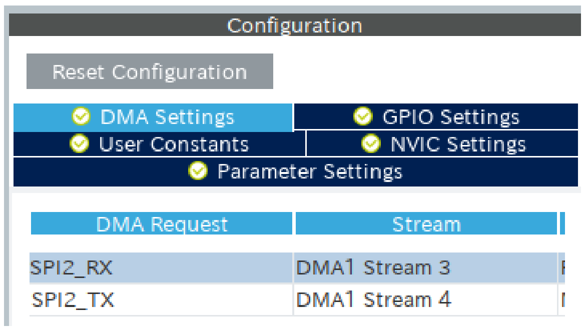
System Core, Analog, Timers, Connectivity: CAN1, CAN2, FMPI2C1, I2C1, I2C2, I2C3, QUADSPI, SDIO, SPI1, SPI2, SPI3, UART4, UART5, USART1, USART2, USART3, USART6, USB_OTG_FS, USB_OTG_HS

Configure the below parameters :

Basic Parameters	
Frame Format	Motorola
Data Size	8 Bits
First Bit	MSB First
Clock Parameters	
Prescaler (for Baud Rate)	2
Baud Rate	22.5 MBits/s
Clock Polarity (CPOL)	Low
Clock Phase (CPHA)	1 Edge
Advanced Parameters	
CRC Calculation	Disabled
NSS Signal Type	Software

SpiMaster class supports the Duplex master, 8bit mode only. The clock phase and Clock polarity is not the matter at the configuration phase.

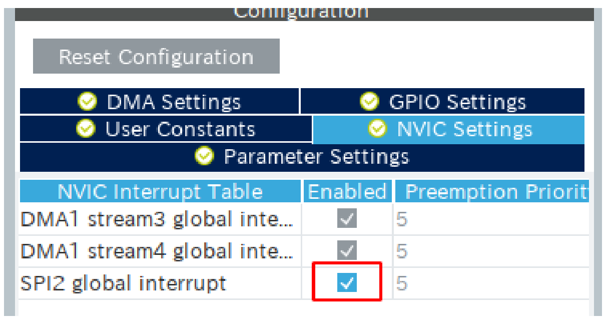
The DMA have to be enabled for both TX and RX. The data size is 8bit for both Peripheral and memory.



Configuration

DMA Request	Stream
SPI2_RX	DMA1 Stream 3
SPI2_TX	DMA1 Stream 4

And then, enable the interrupt.



Configuration

NVIC Interrupt Table	Enabled	Preemption Priorit
DMA1 stream3 global inte...	✓	5
DMA1 stream4 global inte...	✓	5
SPI2 global interrupt	✓	5

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

In addition to the instantiation, we need to prepare an interrupt callback.

```
void HAL_SPI_TxRxCpltCallback (SPI_HandleTypeDef * hspi)
{
    my_spi3->TransmitAndReceiveCompleteCallback(hspi);
}
```

Where HAL_SPI_TxRxCpltCallback is a predefined name of the SPI interrupt handler. This is invoked by system whenever a interrupt baed SPI transmission is complete. Becuase the default function is weakly bound, above definition will overwride the default one.

Note that above callback is invoked for any SPIn where n is 1, 2, 3... To avoid the confusion, SpiMaster::Transfer←CompleteCallback() method chckes whether given parameter matches with its SPI_HandleTypeDef ∗ pointer (which was passed to constructor). And only when both matches, the member function execute the interrupt termination process.

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.

7.32.2 Constructor & Destructor Documentation

7.32.2.1 murasaki::SpiMaster::SpiMaster (SPI_HandleTypeDef ∗ *spi_handle*)

Parameters

spi_handle	Handle to the SPI peripheral. This have to be configured to use DMA by CubeIDE.

7.32.3 Member Function Documentation

7.32.3.1 bool murasaki::SpiMaster::HandleError (void ∗ *ptr*) [virtual]

Parameters

ptr	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 function have to be called from HAL_SPI_ErrorCallback.

```
void HAL_SPI_ErrorCallback(SPI_HandleTypeDef *hspi) {
     if ( murasaki::platform.spi1->HandleError(hspi) )
         return;
}
```

Implements murasaki::SpiMasterStrategy.

7.32.3.2 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

spi_spec	A pointer to the AbstractSpiSpecification to specify the slave device.
tx_data	Data to be transmitted
rx_data	Data buffer to receive data
size	Transfer data size [byte] for each way.
timeout_ms	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 funciton 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.

7.32.3.3 bool murasaki::SpiMaster::TransmitAndReceiveCompleteCallback (void ∗ *ptr*) [virtual]

Parameters

ptr	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::SpiMasterStrategy.

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

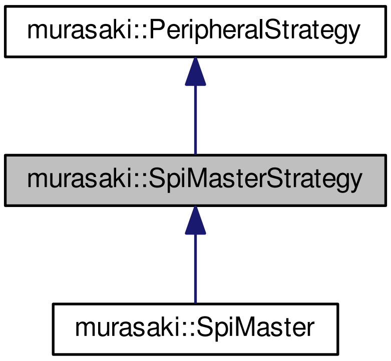
- spimaster.hpp
- spimaster.cpp

7.33 murasaki::SpiMasterStrategy Class Reference

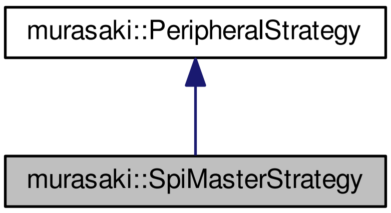
Root class of the SPI master.

`#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.
- virtual bool TransmitAndReceiveCompleteCallback (void ∗ptr)=0

  Callback to notifiy the end of transfer.
- virtual bool HandleError (void ∗ptr)=0

  Handling error report of device.

Additional Inherited Members

7.33.1 Detailed Description

This class provides a thread safe, synchronous SPI transfer.

7.33.2 Member Function Documentation

7.33.2.1 virtual bool murasaki::SpiMasterStrategy::HandleError (void ∗ *ptr*) [pure virtual]

Parameters

ptr	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::SpiMaster.

7.33.2.2 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

spi_spec	Pointer to the SPI slave adapter which has clock configuraiton and chip select handling.
tx_data	Data to be transmitted
rx_data	Data buffer to receive data
size	Transfer data size [byte] for each way. Must be smaller than 65536
timeout_ms	Timeout limit [mS]

Returns

true if transfer complete, false if timeout

Implemented in murasaki::SpiMaster.

7.33.2.3 virtual bool murasaki::SpiMasterStrategy::TransmitAndReceiveCompleteCallback (void ∗ *ptr*) [pure virtual]

Parameters

ptr	Pointer to the control object.

Returns

true if no error.

Implemented in murasaki::SpiMaster.

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

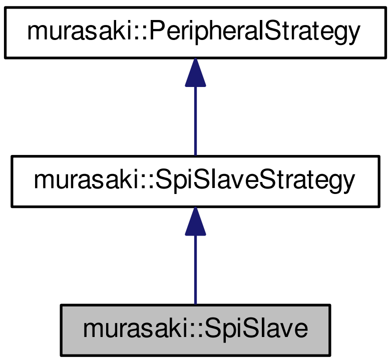
- spimasterstrategy.hpp

7.34 murasaki::SpiSlave Class Reference

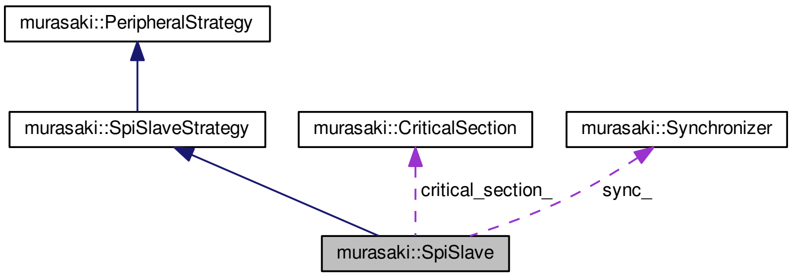
Thread safe, synchronous and blocking IO. Encapsulating SPI slave. Based on STM32Cube HAL driver and Free↩ RTOS.

`#include <spislave.hpp>`

Inheritance diagram for murasaki::SpiSlave:



Collaboration diagram for murasaki::SpiSlave:


murasaki::PeripheralStrategy
murasaki::SpiSlaveStrategy
murasaki::CriticalSection
murasaki::Synchronizer
critical_section_
sync_
murasaki::SpiSlave

Public Member Functions

- SpiSlave (SPI_HandleTypeDef ∗spi_handle)

  Constractor.
- virtual SpiStatus TransmitAndReceive (const uint8_t ∗tx_data, uint8_t ∗rx_data, unsigned int size, unsigned int ∗transfered_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.

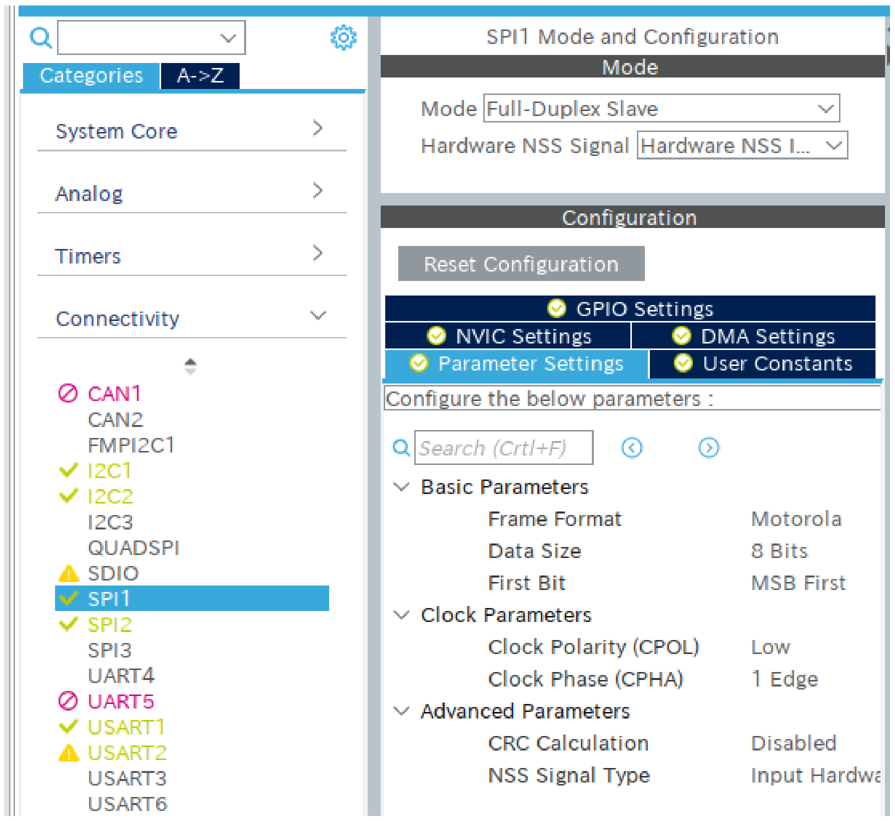
Additional Inherited Members

7.34.1 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



Mode Full-Duplex Slave
Hardware NSS Signal Hardware NSS I...

Configure the below parameters :

Basic Parameters

Frame Format	Motorola
Data Size	8 Bits
First Bit	MSB First

Clock Parameters

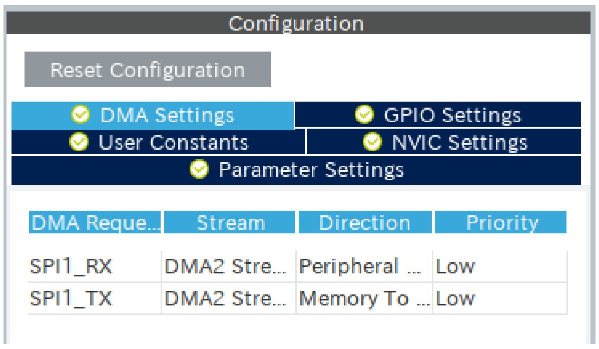
Clock Polarity (CPOL)	Low
Clock Phase (CPHA)	1 Edge

Advanced Parameters

CRC Calculation	Disabled
NSS Signal Type	Input Hardwa

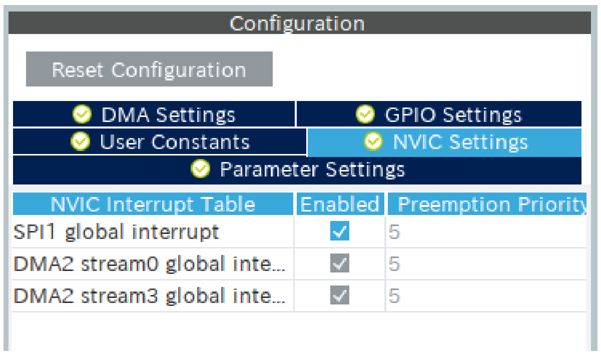
SpiMaster class supports the Duplex slave, 8bit mode only. Also the clock phase and Clock polarity have to be configured by the configuration tool. This is different from master.

The DMA have to be enabled for both TX and RX. The data size is 8bit for both Peripheral and memory.



DMA Reque...	Stream	Direction	Priority
SPI1_RX	DMA2 Stre...	Peripheral ...	Low
SPI1_TX	DMA2 Stre...	Memory To ...	Low

And then, enable the interrupt.



NVIC Interrupt Table	Enabled	Preemption Priority
SPI1 global interrupt	✓	5
DMA2 stream0 global inte...	✓	5
DMA2 stream3 global inte...	✓	5

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

To use this class, the SPI peripheral have to be configured to use the interrupt and DMA. Also the bitrate, CPOL and CPHA should be configured by the CubeIDE.

In addition to the instantiation, we need to prepare an interrupt callback.

```
void HAL_SPI_TxRxCpltCallback (SPI_HandleTypeDef * hspi)
{
    my_spi3->TransmitAndReceiveCompleteCallback(hspi);
}
```

Where HAL_SPI_TxRxCpltCallback is a predefined name of the SPI interrupt handler. This is invoked by system whenever a interrupt baed SPI transmission is complete. Because the default function is weakly bound, above definition will override the default one.

Note that above callback is invoked for any SPIn where n is 1, 2, 3... To avoid the confusion, SpiSlave::Transfer↩CompleteCallback() method checkes whether given parameter matches with its SPI_HandleTypeDef ∗ pointer (which was passed to constructor). And only when both matches, the member function execute the interrupt termination process.

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.

7.34.2 Constructor & Destructor Documentation

7.34.2.1 murasaki::SpiSlave::SpiSlave (SPI_HandleTypeDef ∗ *spi_handle*)

Parameters

spi_handle	Handle to the SPI peripheral. This have to be configured to use DMA by CubeIDE.

7.34.3 Member Function Documentation

7.34.3.1 bool murasaki::SpiSlave::HandleError (void ∗ *ptr*) [virtual]

Parameters

ptr	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_SPI_ErrorCallback()

```
void HAL_SPI_ErrorCallback(SPI_HandleTypeDef * hspi) {
     if ( murasaki::platform.spi1->HandleError(hspi) )
         return;
}
```

Implements murasaki::SpiSlaveStrategy.

7.34.3.2 SpiStatus murasaki::SpiSlave::TransmitAndReceive (const uint8_t ∗ *tx_data,* uint8_t ∗ *rx_data,* unsigned int *size,* unsigned int ∗ *transfered_count,* unsigned int *timeout_ms* = murasaki::kwmsIndefinitely) [virtual]

Parameters

tx_data	Data to be transmitted
rx_data	Data buffer to receive data
size	Transfer data size [byte] for each way.
transfered_count	(Currently, Just ignored) The transfered number of bytes during API.
timeout_ms	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 funciton 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.

7.34.3.3 bool murasaki::SpiSlave::TransmitAndReceiveCompleteCallback (void ∗ *ptr*) [virtual]

Parameters

ptr	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.spi1->TransmitAndReceiveCompleteCallback(hspi))
        return;
}
```

Implements murasaki::SpiSlaveStrategy.

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

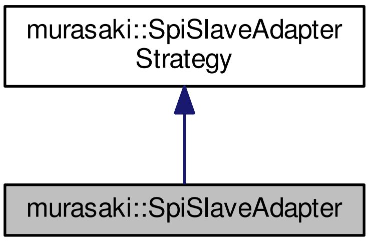
- spislave.hpp
- spislave.cpp

7.35 murasaki::SpiSlaveAdapter Class Reference

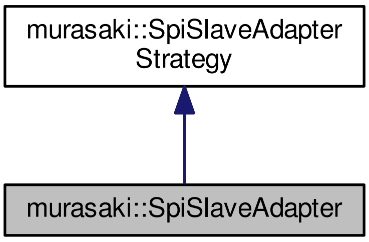
A speficier of SPI slave.

```
#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

7.35.1 Detailed Description

This class describes how ths slave is. The description is clock POL and PHA for the speicific 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.

7.35.2 Constructor & Destructor Documentation

7.35.2.1 murasaki::SpiSlaveAdapter::SpiSlaveAdapter (murasaki::SpiClockPolarity *pol*, murasaki::SpiClockPhase *pha*, ::GPIO_TypeDef * *port*, uint16_t *pin*)

Parameters

pol	Polarity setting
pha	Phase setting
port	GPIO port of the chip select
pin	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.

7.35.2.2 murasaki::SpiSlaveAdapter::SpiSlaveAdapter (unsigned int *pol*, unsigned int *pha*, ::GPIO_TypeDef *const *port*, uint16_t *pin*)

Parameters

pol	Polarity setting
pha	Phase setting
port	GPIO port of the chip select
pin	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.

7.35.3 Member Function Documentation

7.35.3.1 void murasaki::SpiSlaveAdapter::AssertCs ( ) [virtual]

This member function asset the output line to select the slave chip.

Reimplemented from murasaki::SpiSlaveAdapterStrategy.

7.35.3.2 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:

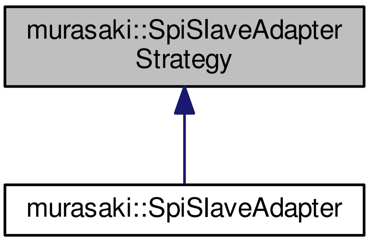
- spislaveadapter.hpp
- spislaveadapter.cpp

7.36 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.

7.36.1 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.

7.36.2 Constructor & Destructor Documentation

7.36.2.1 murasaki::SpiSlaveAdapterStrategy::SpiSlaveAdapterStrategy (**murasaki::SpiClockPolarity** *pol*, **murasaki::SpiClockPhase** *pha*)

Parameters

pol	Polarity setting
pha	Phase setting

7.36.2.2 murasaki::SpiSlaveAdapterStrategy::SpiSlaveAdapterStrategy (unsigned int *pol*, unsigned int *pha*)

Parameters

pol	Polarity setting
pha	Phase setting

7.36.3 Member Function Documentation

7.36.3.1 void murasaki::SpiSlaveAdapterStrategy::AssertCs ( ) `[virtual]`

This member function asset the output line to select the slave chip.

This have to be overriden.

Reimplemented in murasaki::SpiSlaveAdapter.

7.36.3.2 void murasaki::SpiSlaveAdapterStrategy::DeassertCs ( ) `[virtual]`

This member function deasset the output line to de-select the slave chip.

This have to be overriden.

Reimplemented in murasaki::SpiSlaveAdapter.

7.36.3.3 murasaki::SpiClockPhase murasaki::SpiSlaveAdapterStrategy::GetCpha ( )

Returns

CPHA setting

7.36.3.4 murasaki::SpiClockPolarity murasaki::SpiSlaveAdapterStrategy::GetCpol ( )

Returns

CPOL setting

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

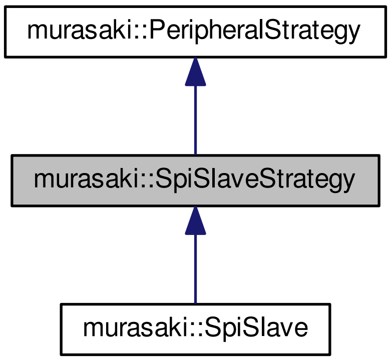
- spislaveadapterstrategy.hpp
- spislaveadapterstrategy.cpp

7.37 murasaki::SpiSlaveStrategy Class Reference

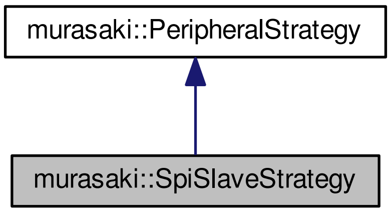
Root class of the SPI slave.

```
#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 ∗transfered_count=nullptr, unsigned int timeout_ms=murasaki::kwmsIndefinitely)=0

    Thread safe, synchronous SPI transfer.
- virtual bool TransmitAndReceiveCompleteCallback (void ∗ptr)=0

    Callback to notifiy the end of transfer.
- virtual bool HandleError (void ∗ptr)=0

    Handling error report of device.

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 bool murasaki::SpiSlaveStrategy::HandleError (void ∗ *ptr*) `[pure virtual]`

Parameters

ptr	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.

7.37.2.2 virtual SpiStatus murasaki::SpiSlaveStrategy::TransmitAndReceive (const uint8_t ∗ *tx_data,* uint8_t ∗ *rx_data,* unsigned int *size,* unsigned int ∗ *transfered_count* = `nullptr`, unsigned int *timeout_ms* = murasaki::kwmsIndefinitely) `[pure virtual]`

Parameters

tx_data	Data to be transmitted
rx_data	Data buffer to receive data
size	Transfer data size [byte] for each way. Must be smaller than 65536
transfered_count	The transfered number of bytes during API.
timeout_ms	Timeout limit [mS]

Returns

true if transfer complete, false if timeout

Implemented in murasaki::SpiSlave.

7.37.2.3 virtual bool murasaki::SpiSlaveStrategy::TransmitAndReceiveCompleteCallback (void ∗ *ptr*) `[pure virtual]`

Parameters

ptr	Pointer to the control object.

Returns

true if no error.

Implemented in murasaki::SpiSlave.

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

- spislavestrategy.hpp

7.38 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. Asigning 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.

7.38.1 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.

7.38.2 Member Function Documentation

7.38.2.1 void murasaki::Synchronizer::Release ( )

Release the task waiting. This member function can be called from both task and the interrupt context.

7.38.2.2 bool murasaki::Synchronizer::Wait (unsigned int *timeout_ms* = **kwmsIndefinitely**)

Parameters

timeout_ms	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:

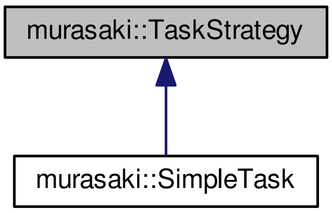
- synchronizer.hpp
- synchronizer.cpp

7.39 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()

7.39.1 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 thask 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/`

7.39.2 Constructor & Destructor Documentation

7.39.2.1 murasaki::TaskStrategy::TaskStrategy (const char ∗ *task_name*, unsigned short *stack_depth*, murasaki::TaskPriority *task_priority*, const void ∗ *task_parameter*)

Parameters

task_name	Name of task. Will be passed to task when started.
stack_depth	[Byte]
task_priority	Priority of the task. from 1 to up to configMAX_PRIORITIES -1. The high number is the high priority.
task_parameter	Optional parameter to the task.

7.39.3 Member Function Documentation

7.39.3.1 const char ∗ murasaki::TaskStrategy::GetName ( )

Returns

A name of task.

7.39.3.2 unsigned int murasaki::TaskStrategy::getStackDepth ( )

Returns

Total depth of the task stack [byte]

7.39.3.3 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.

7.39.3.4 void murasaki::TaskStrategy::Launch (void * *ptr*) `[static]`, `[protected]`

Parameters

ptr	passing "this" pointer.

7.39.3.5 void murasaki::TaskStrategy::Start (void)

A task is created with given parameter to the constructors and then run.

7.39.3.6 virtual void murasaki::TaskStrategy::TaskBody (const void * *ptr*) `[protected]`, `[pure virtual]`

Parameters

ptr	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:

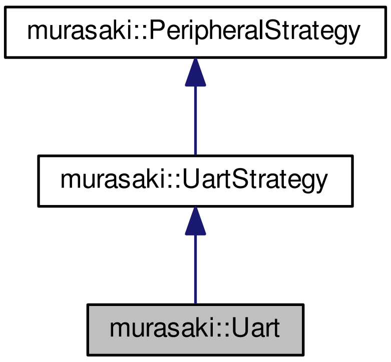
- taskstrategy.hpp
- taskstrategy.cpp

7.40 murasaki::Uart Class Reference

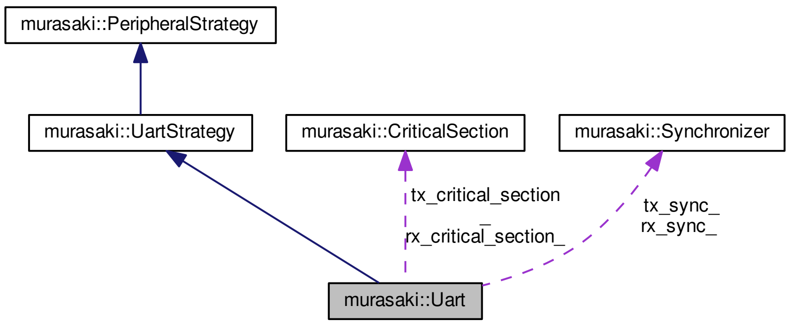
Thread safe, synchronous and blocking IO. Concrete implementation of UART controller. Based on the STM32Cube HAL DMA Transfer.

```
#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 ∗transfered_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.

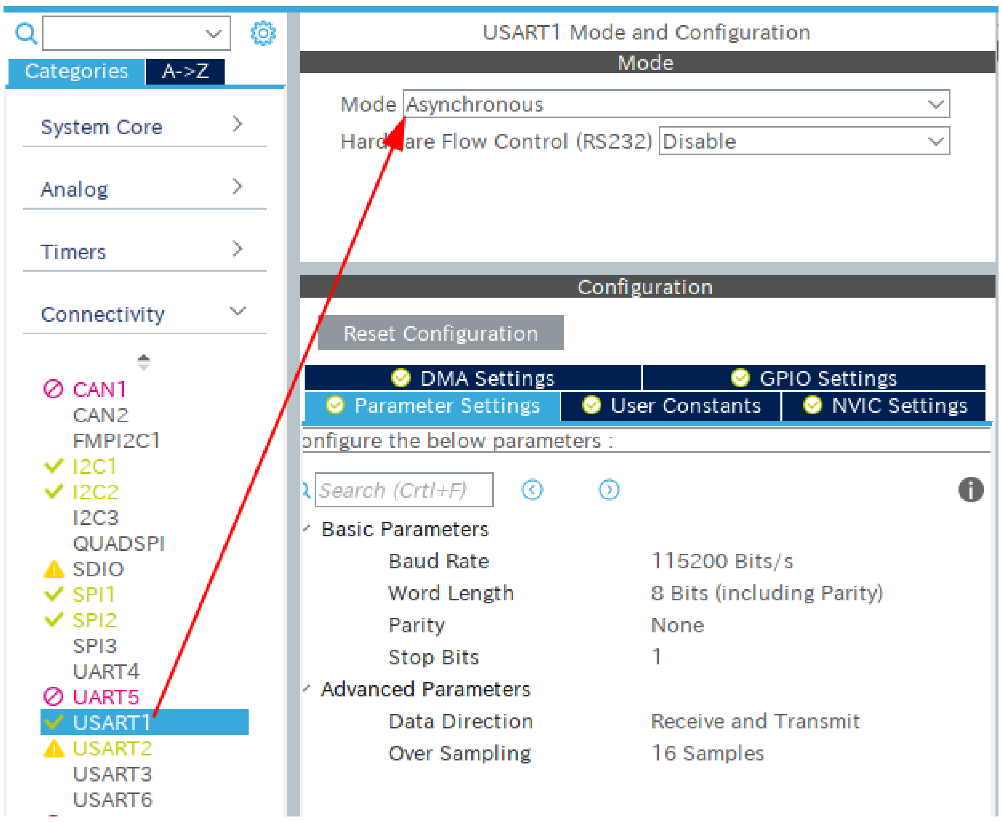
Additional Inherited Members

7.40.1 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.

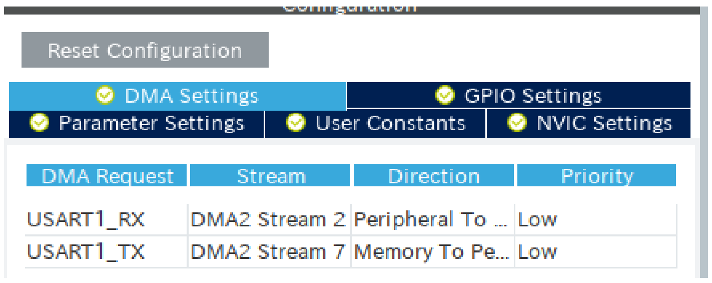


Mode: Asynchronous
Hardware Flow Control (RS232): Disable

Basic Parameters	
Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Advanced Parameters	
Data Direction	Receive and Transmit
Over Sampling	16 Samples

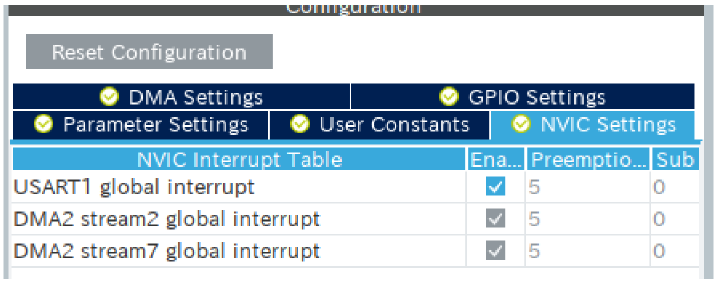
Make sure setting direction to Receive and Transmit. Other parameters are up to the application.

The DMA have to be enabled for both TX and RX. The data size is 8bit for both Peripheral and memory.



DMA Request	Stream	Direction	Priority
USART1_RX	DMA2 Stream 2	Peripheral To ...	Low
USART1_TX	DMA2 Stream 7	Memory To Pe...	Low

And then, enable the interrupt.



NVIC Interrupt Table	Ena...	Preemptio...	Sub
USART1 global interrupt	✓	5	0
DMA2 stream2 global interrupt	✓	5	0
DMA2 stream7 global interrupt	✓	5	0

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

In addition to the instantiation, we need to prepare an interrupt callback.

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef * huart)
{
    if (my_uart3->TransmitCompleteCallback(huart))
        return;
}
```

Where HAL_UART_TxCpltCallback is a predefined name of the UART interrupt handler. This is invoked by system whenever a DMA baed UART transmission is complete. Becuase the default function is weakly bound, above definition will overwride the default one.

Note that above callback is invoked for any UARTn where n is 1, 2, 3... To avoid the confusion, Uart::Transmit↩ CompleteCallback() method chckes whether given parameter matches with its UART_HandleTypeDef ∗ pointer (which was passed to constructor). And only when both matches, the member function execute the interrupt termination process.

As same as Tx, RX needs HAL_UART_TxCpltCallback().

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 specifes 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 specifes 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.

7.40.2 Constructor & Destructor Documentation

7.40.2.1 murasaki::Uart::Uart (UART_HandleTypeDef ∗ *uart*)

Parameters

uart	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.

7.40.3 Member Function Documentation

7.40.3.1 bool murasaki::Uart::HandleError (void ∗const *ptr*) `[virtual]`

Parameters

ptr	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 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.

7.40.3.2 murasaki::UartStatus murasaki::Uart::Receive (uint8_t ∗ *data,* unsigned int *count,* unsigned int ∗ *transfered_count,* UartTimeout *uart_timeout,* unsigned int *timeout_ms*) `[virtual]`

Parameters

data	Data buffer to place the received data..
count	The count of the data (byte) to be transfered. Must be smaller than 65536
transfered_count	(Currently, Just ignored) Number of bytes transfered. The nullPtr means no need to return value.
uart_timeout	Specify murasaki::kutIdleTimeout, if idle line timeout is needed.
timeout_ms	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 retun 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. Periperal 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.

7.40.3.3 bool murasaki::Uart::ReceiveCompleteCallback (void ∗const *ptr*) `[virtual]`

Parameters

ptr	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.

7.40.3.4 void murasaki::Uart::SetHardwareFlowControl (UartHardwareFlowControl *control*) [virtual]

Parameters

control	The control mode.

Before calling this method, all transmission and recevie activites have to be finished. This is responsibility of the programmer.

Note this method is NOT re-etnrant. In other word, this member function can be called from both task and interrupt context.

Reimplemented from murasaki::UartStrategy.

7.40.3.5 void murasaki::Uart::SetSpeed (unsigned int *baud_rate*) [virtual]

Parameters

baud_rate	BAUD rate (110, 300,... 57600,...)

Before calling this method, all transmission and recevie activites have to be finished. This is responsibility of the programmer.

Note this method is NOT re-etnrant. In other word, this member function can be called from both task and interrupt context.

Reimplemented from murasaki::UartStrategy.

7.40.3.6 murasaki::UartStatus murasaki::Uart::Transmit (const uint8_t * *data,* unsigned int *size,* unsigned int *timeout_ms*) [virtual]

Parameters

data	Data buffer to be transmitted.
size	The count of the data (byte) to be transfered. Must be smaller than 65536
timeout_ms	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 forbiddedn to call from ISR.

Implements murasaki::UartStrategy.

7.40.3.7 bool murasaki::Uart::TransmitCompleteCallback (void ∗const *ptr*) [virtual]

Parameters

ptr	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 retun 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:

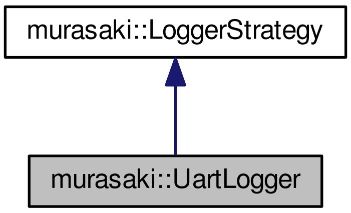
- uart.hpp
- uart.cpp

7.41 murasaki::UartLogger Class Reference

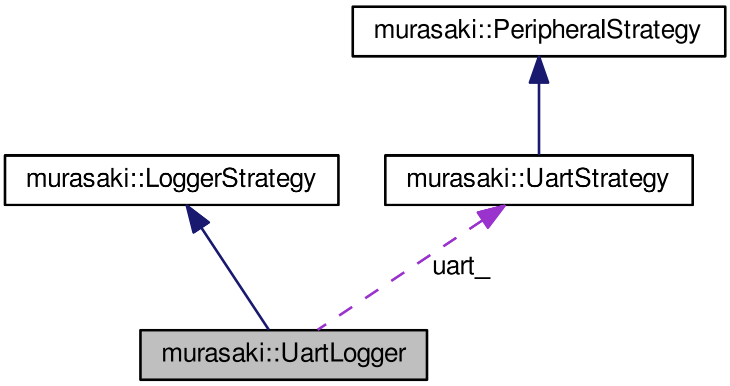
Logging through an UART port.

```
#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

7.41.1 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.

7.41.2 Constructor & Destructor Documentation

7.41.2.1 murasaki::UartLogger::UartLogger (UartStrategy ∗ *uart*)

Parameters

uart	Pointer to the uart object.

7.41.3 Member Function Documentation

7.41.3.1 void murasaki::UartLogger::DoPostMortem (void ∗ *debugger_fifo*) [virtual]

Parameters

debugger_fifo	Pointer to the DebuggerFifo 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 funciton call the DebuggerFifo::SetPostMortem() intenally. 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 funciton rewind the FIFO and then, start to transmit the data again.

Reimplemented from murasaki::LoggerStrategy.

7.41.3.2 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.

7.41.3.3 void murasaki::UartLogger::putMessage (char *message[ ]*, unsigned int *size*) [virtual]

Parameters

message	Non null terminated character array. This data is stored or output to the logger.
size	Size of the message[bytes]. Must be smaller than 65536

Implements murasaki::LoggerStrategy.

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

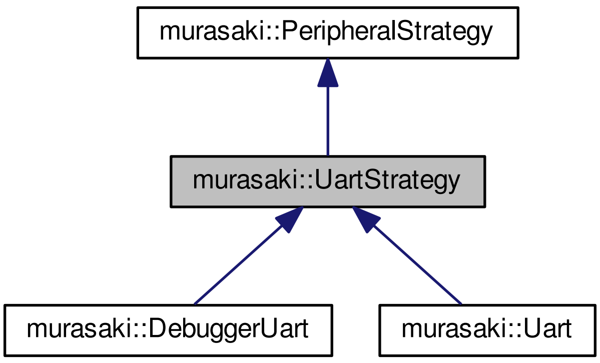
- uartlogger.hpp
- uartlogger.cpp

7.42 murasaki::UartStrategy Class Reference

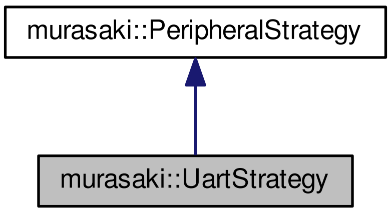
Definition of the root class of UART.

```
#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 ∗transfered_↩ count=nullptr, UartTimeout uart_timeout=murasaki::kutNoIdleTimeout, 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

7.42.1 Detailed Description

A prototype of the UART device. Ths 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 method should be synchronous. That men, until the transmit / receve terminates, both method doesn't return.

Two call back methods are prepared to sync with the interrutp which tells the end of Transmit/Recieve.

7.42.2 Member Function Documentation

7.42.2.1 virtual bool murasaki::UartStrategy::HandleError (void ∗ *ptr*) `[pure virtual]`

Parameters

ptr	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.

7.42.2.2 virtual murasaki::UartStatus murasaki::UartStrategy::Receive (uint8_t ∗ *data*, unsigned int *size*, unsigned int ∗ *transfered_count* = `nullptr`, UartTimeout *uart_timeout* = murasaki::kutNoIdleTimeout, unsigned int *timeout_ms* = murasaki::kwmsIndefinitely) `[pure virtual]`

Parameters

data	Pointer to the buffer to save the received data.
size	Number of the data to be received.
transfered_count	Number of bytes transfered. The nullPtr means no need to return value.
uart_timeout	Specify murasaki::kutIdleTimeout, if idle line timeout is needed.
timeout_ms	Time out by milli Second.

Returns

Status of the IO processing

Implemented in murasaki::Uart, and murasaki::DebuggerUart.

7.42.2.3 virtual bool murasaki::UartStrategy::ReceiveCompleteCallback (void ∗ *ptr*) [pure virtual]

Parameters

ptr	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.

7.42.2.4 virtual void murasaki::UartStrategy::SetHardwareFlowControl (UartHardwareFlowControl *control*) [inline], [virtual]

Parameters

control	The control mode.

Reimplemented in murasaki::Uart, and murasaki::DebuggerUart.

7.42.2.5 virtual void murasaki::UartStrategy::SetSpeed (unsigned int *speed*) [inline], [virtual]

Parameters

speed	BAUD rate (110, 300, ... 9600,...)

Reimplemented in murasaki::Uart, and murasaki::DebuggerUart.

7.42.2.6 virtual **murasaki::UartStatus** murasaki::UartStrategy::Transmit (const uint8_t ∗ *data*, unsigned int *size*, unsigned int *timeout_ms* = **murasaki::kwmsIndefinitely**) [pure virtual]

Parameters

data	Pointer to the buffer to be sent.
size	Number of the data to be sent.
timeout_ms	Time out by mili Second.

Returns

Status of the IO processing

Implemented in murasaki::Uart, and murasaki::DebuggerUart.

7.42.2.7 virtual bool murasaki::UartStrategy::TransmitCompleteCallback (void ∗ *ptr*) [pure virtual]

Parameters

ptr	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:

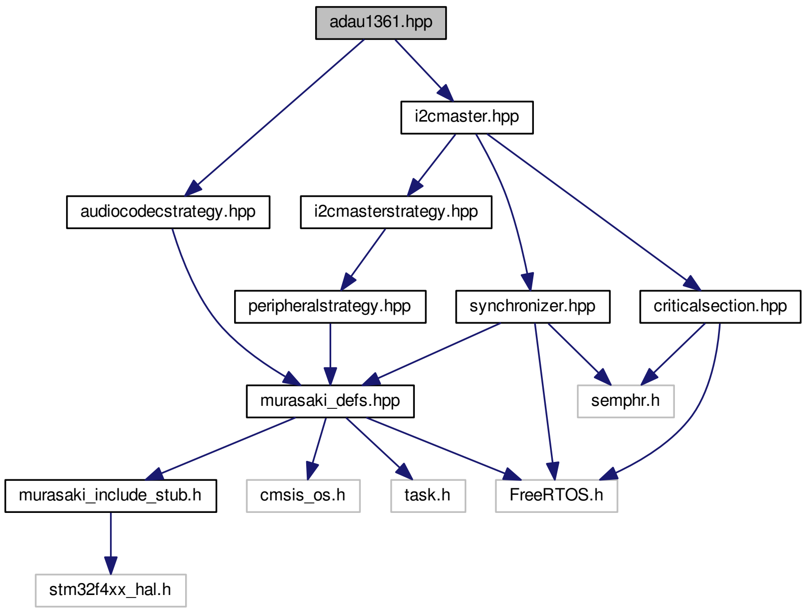
- uartstrategy.hpp

8 File Documentation

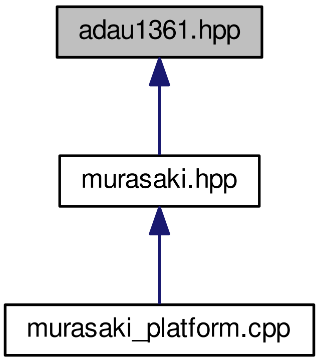
8.1 adau1361.hpp File Reference

```
#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

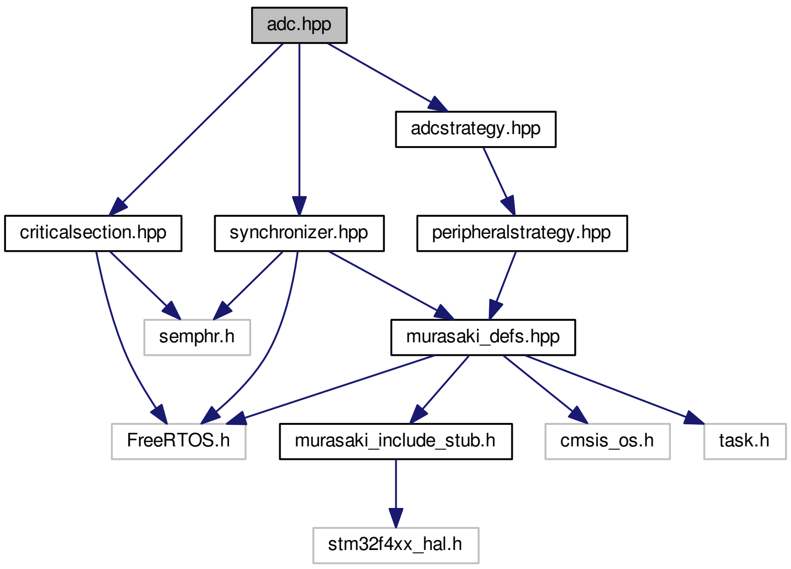
: Seiichi "Suikan" Horie

8.2 adc.hpp File Reference

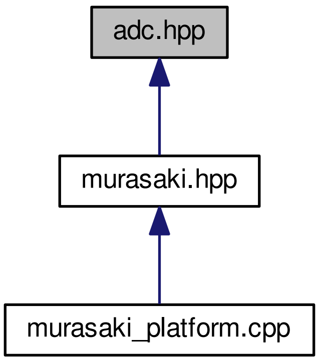
AD Coverter class.

```
#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

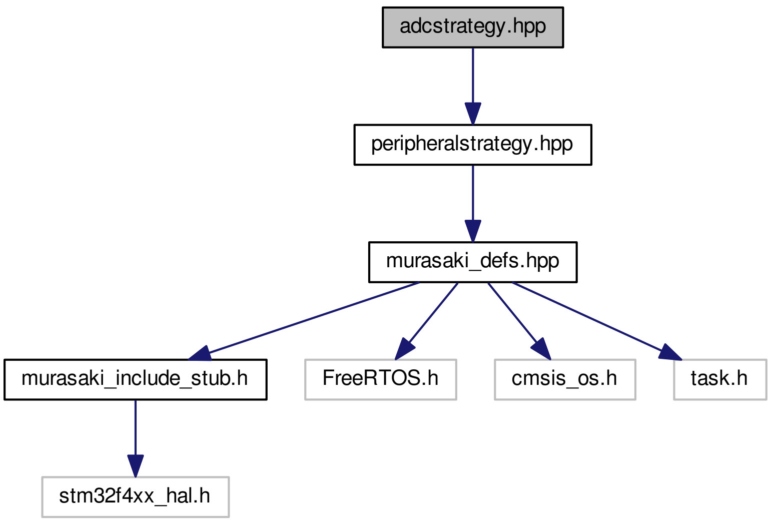
Author

Seiichi Horie

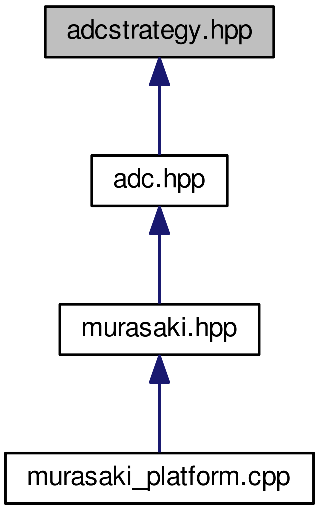
8.3 adcstrategy.hpp File Reference

```
#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

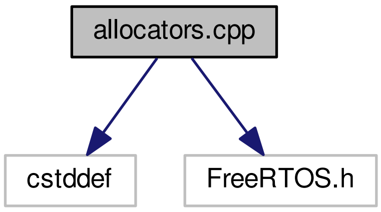
Author

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

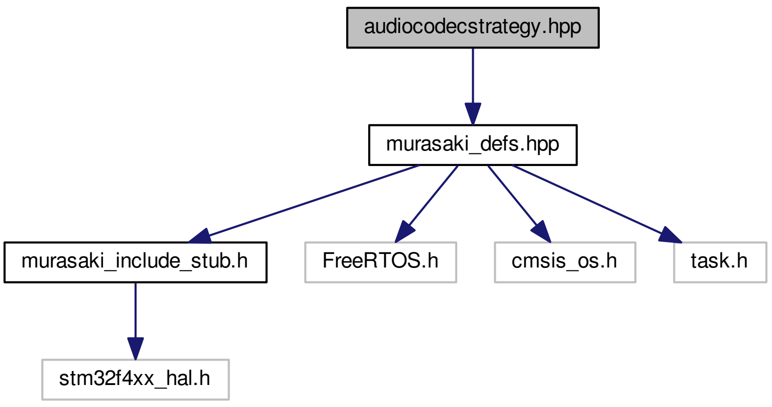
These definitions allows to used the FreeRTOS's heap instead of the system heap.

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 commiting memory.

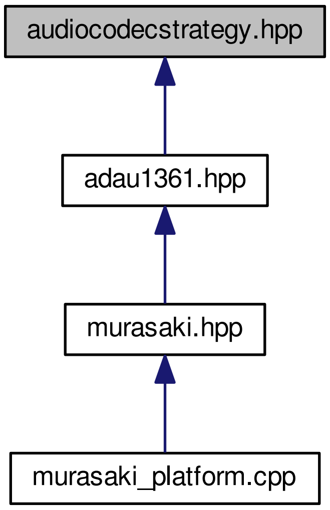
FreeRTOS hepa is considered safer than system heap. Then, the new and the delete operators are overloaded to use the pvPortMalloc().

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

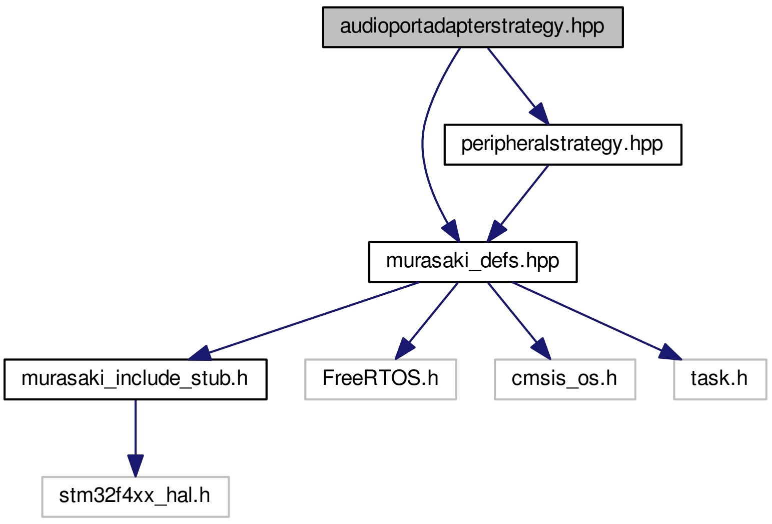
: Seiichi "Suikan" Horie

8.6 audioportadapterstrategy.hpp File Reference

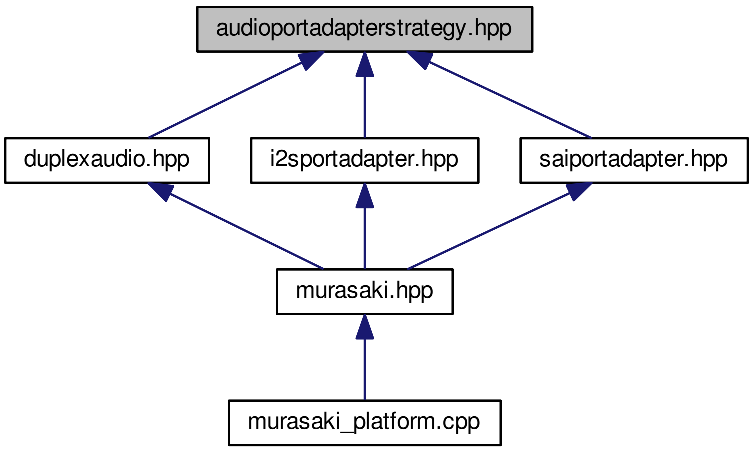
Strategy of the Audio device adaptor.

```
#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

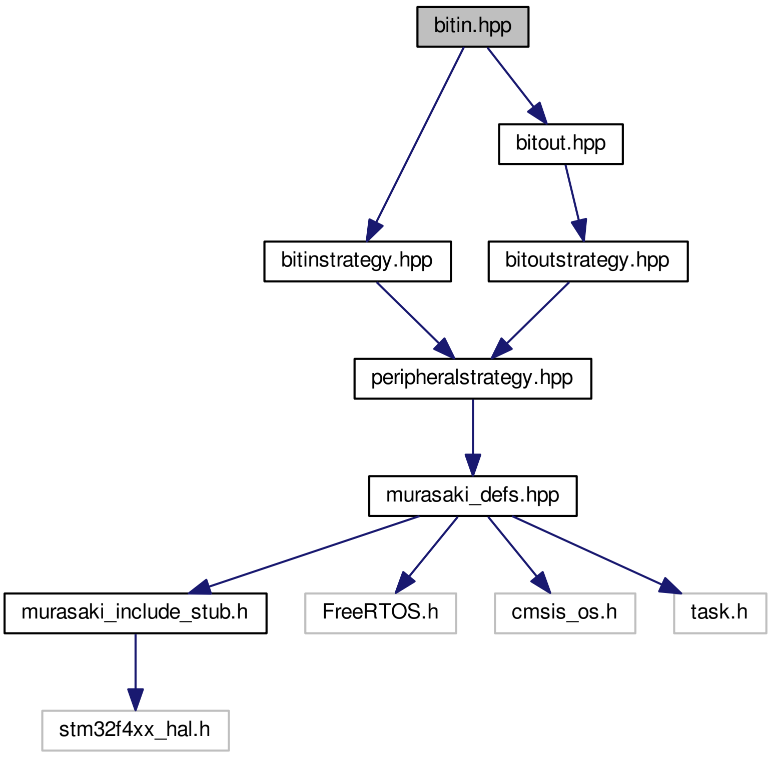
Seiichi "Suikan" Horie

8.7 bitin.hpp File Reference

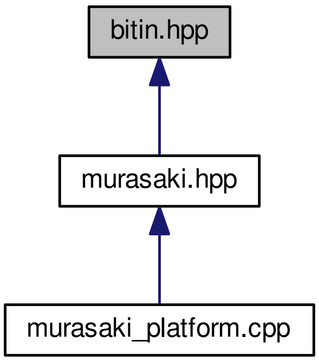
GPIO bit in class.

```
#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

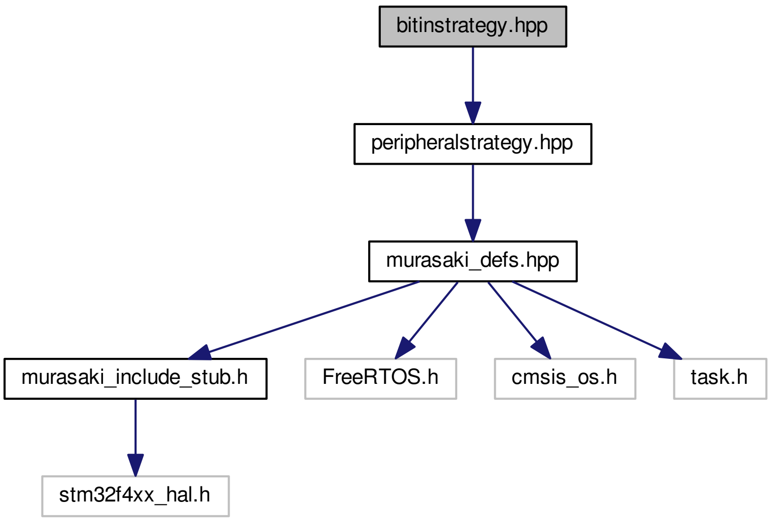
Seiichi "Suikan" Horie

8.8 bitinstrategy.hpp File Reference

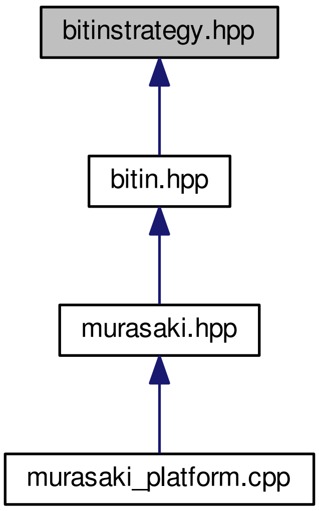
Abstract class of the GPIO bit in.

```
#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

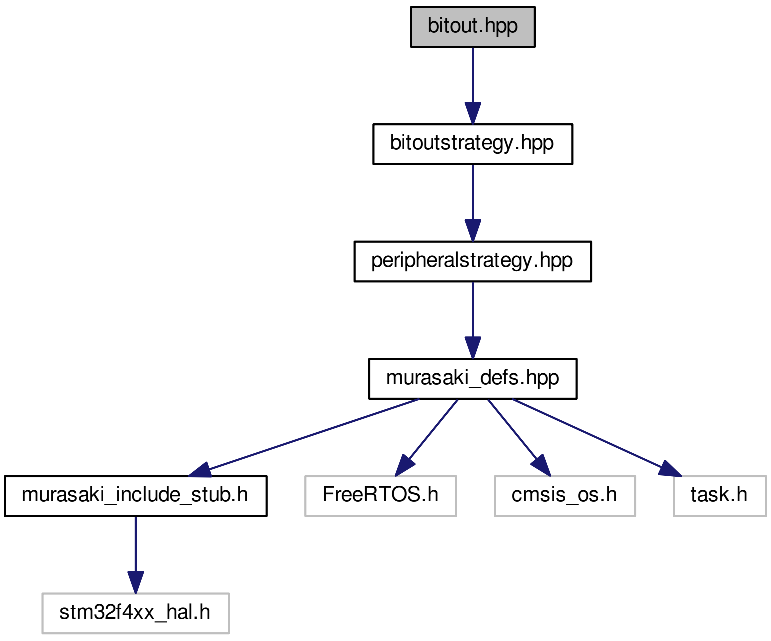
Seiichi "Suikan" Horie

8.9 bitout.hpp File Reference

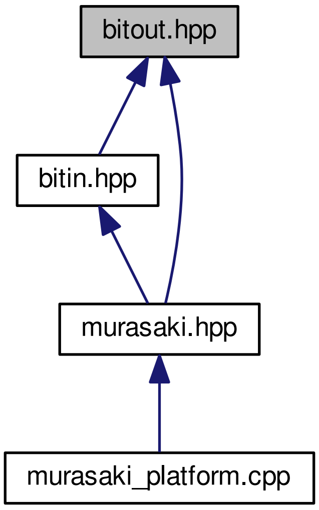
GPIO bit out class.

```
#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

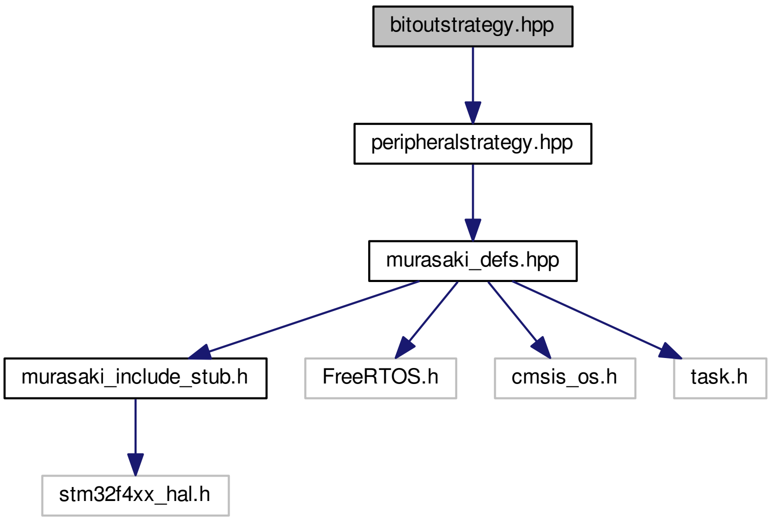
Seiichi "Suikan" Horie

8.10 bitoutstrategy.hpp File Reference

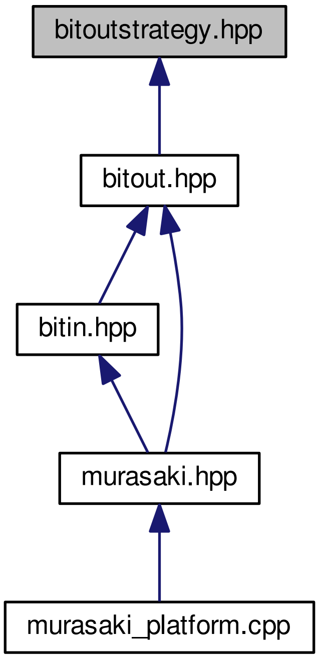
Abstract class of GPIO bit out.

```
#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

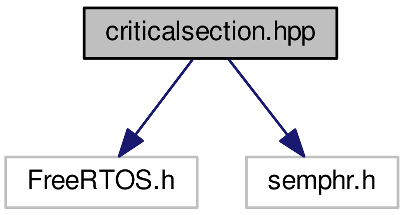
Seiichi "Suikan" Horie

8.11 criticalsection.hpp File Reference

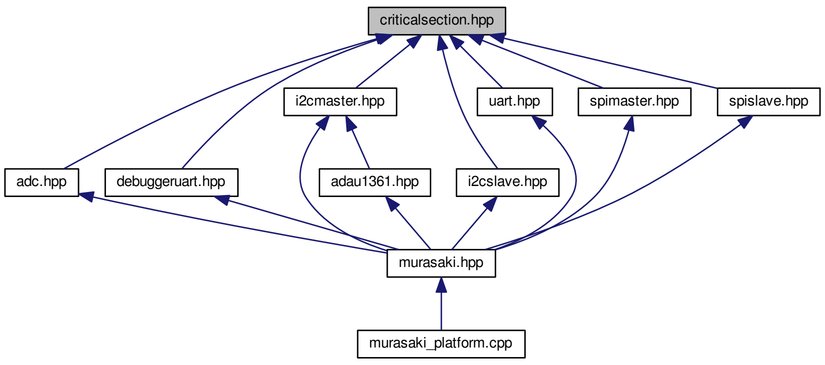
Class to protect a certain section from the interference.

```
#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.

8.11.1 Detailed Description

Date

2018/01/27

Author

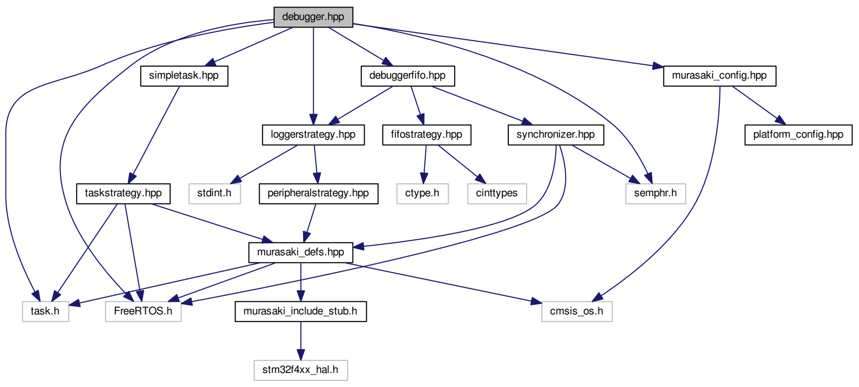
Seiichi "Suikan" Horie

8.12 debugger.hpp File Reference

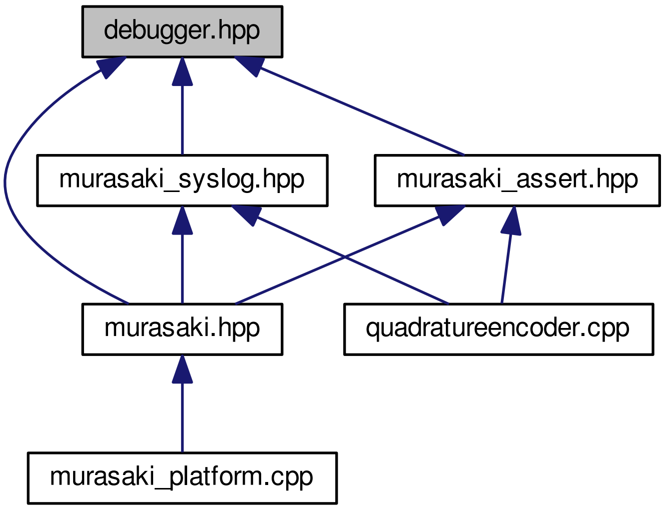
Debug print class. For both ISR and task.

```
#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.

8.12.1 Detailed Description

Date

2018/01/03

Author

Seiichi "Suikan" Horie

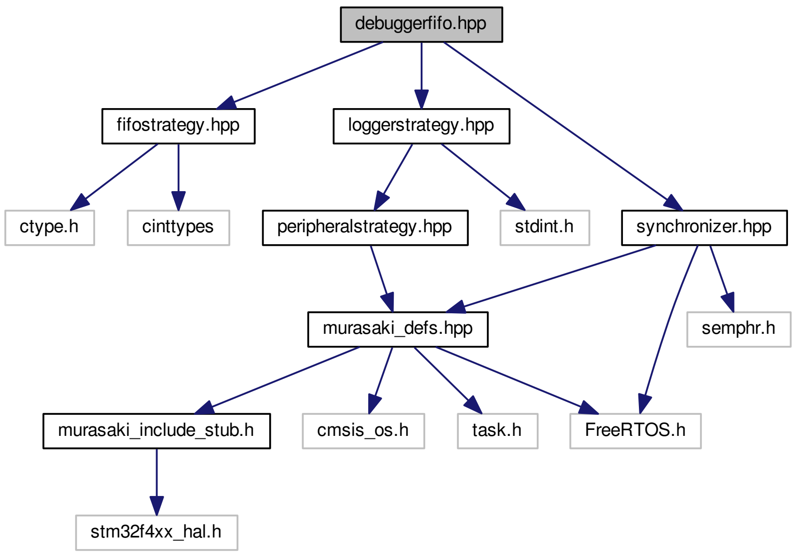
This class serves printf function for both task context and ISR context.

8.13 debuggerfifo.hpp File Reference

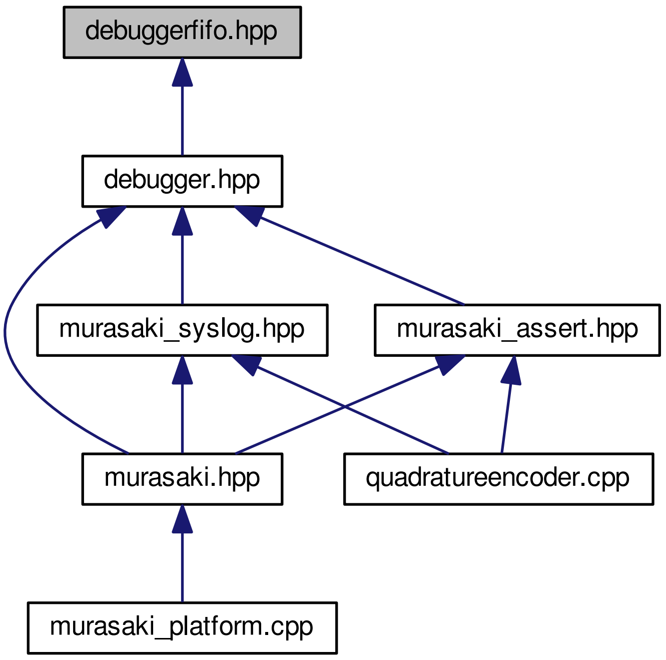
Dedicated FIFO to logging the debug message.

```
#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.

8.13.1 Detailed Description

Date

2018/03/01

Author

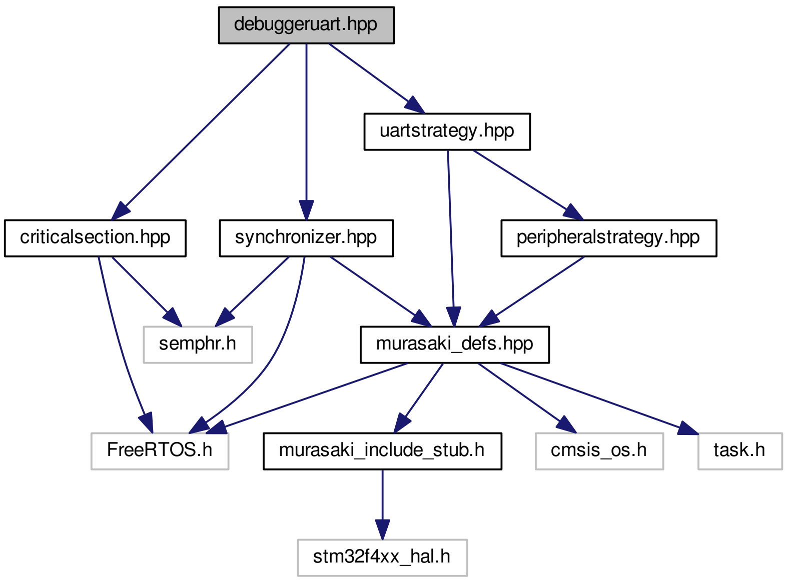
Seiichi "Suikan" Horie

8.14 debuggeruart.hpp File Reference

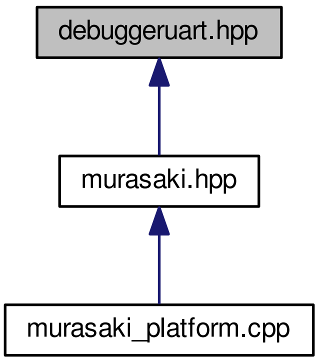
UART. Thread safe and synchronous IO.

```
#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.

8.14.1 Detailed Description

Date

2018/09/23

Author

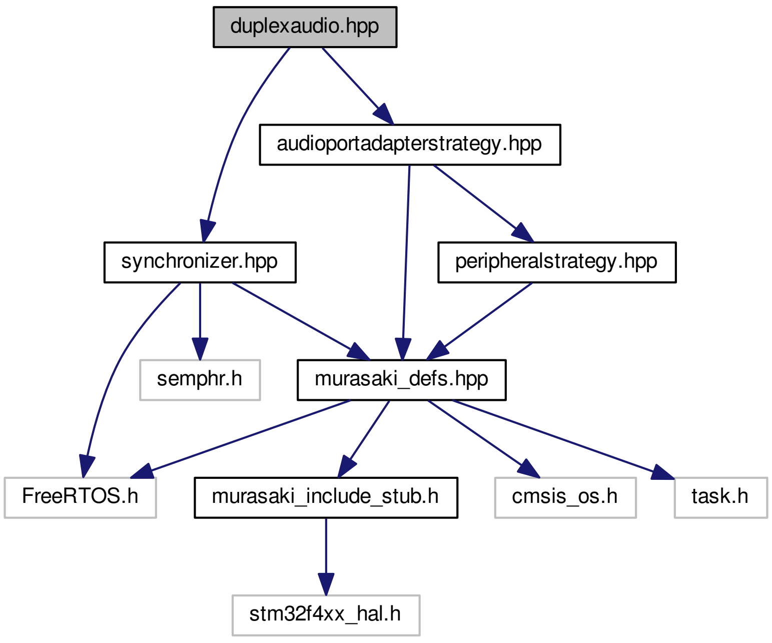
Seiichi "Suikan" Horie

8.15 duplexaudio.hpp File Reference

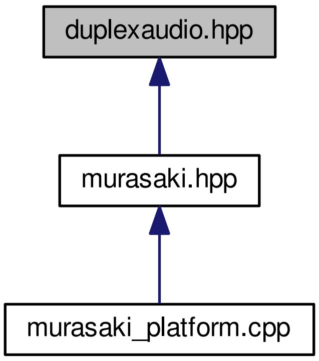
root class of the stereo audio

```
#include <synchronizer.hpp>
#include "audioportadapterstrategy.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 the descendants of this class.

Namespaces

- murasaki

  Personal Platform parts collection.

8.15.1 Detailed Description

Date

2019/03/02

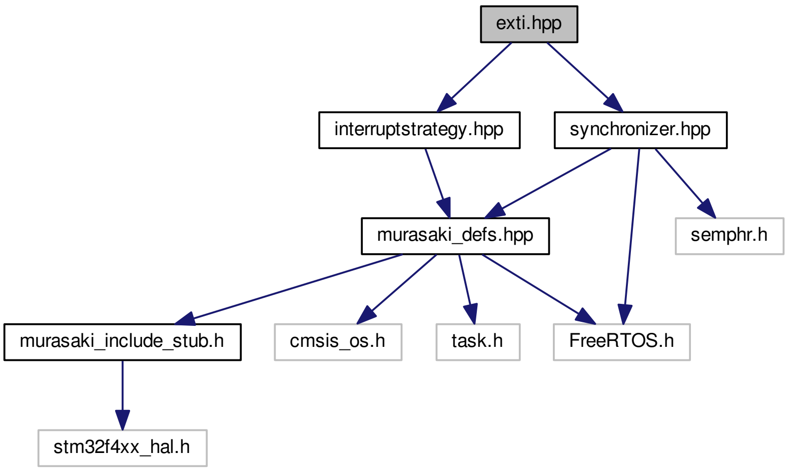
Author

Seiichi "Suikan" Horie

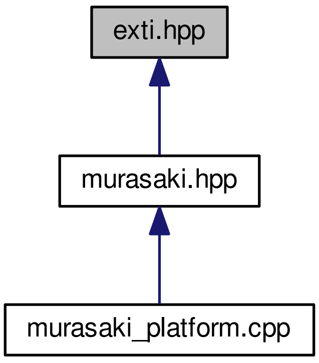
8.16 exti.hpp File Reference

```
#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.

8.16.1 Detailed Description

Date

Feb 29, 2020

Author

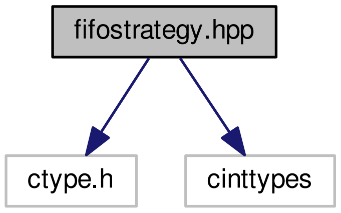
takemasa

8.17 fifostrategy.hpp File Reference

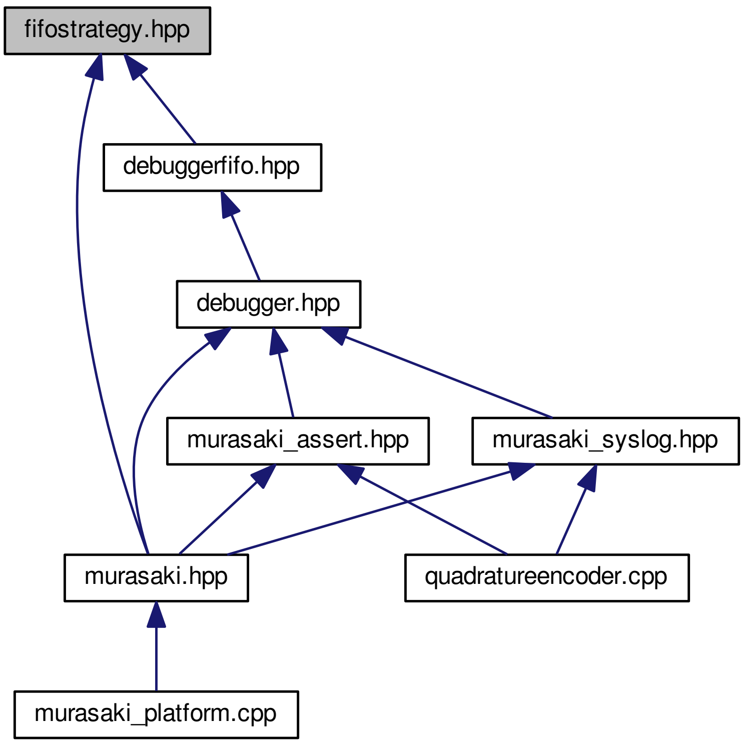
Abstract class of FIFO.

```
#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.

8.17.1 Detailed Description

Date

2018/02/26

Author

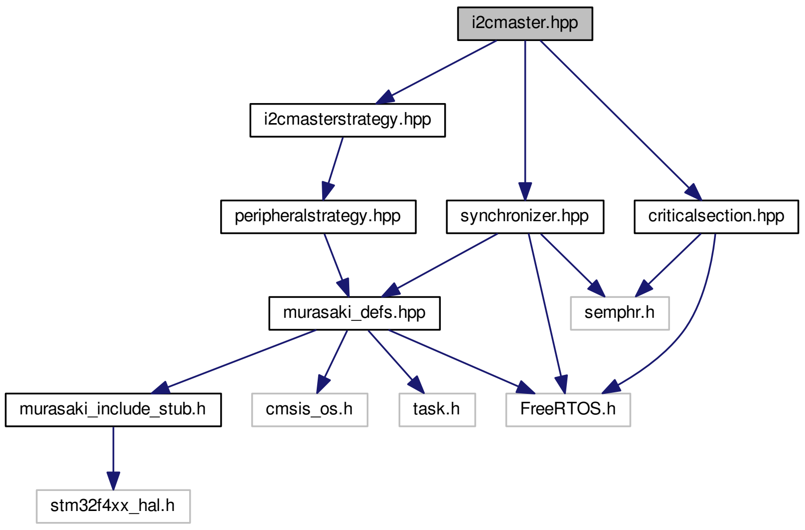
Seiichi "Suikan" Horie

8.18 i2cmaster.hpp File Reference

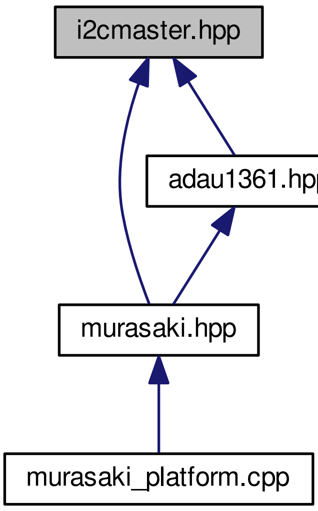
I2C master. Thread safe, synchronous IO.

```
#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 Free↩ RTOS.

Namespaces

- murasaki

  Personal Platform parts collection.

8.18.1 Detailed Description

Date

2018/02/12

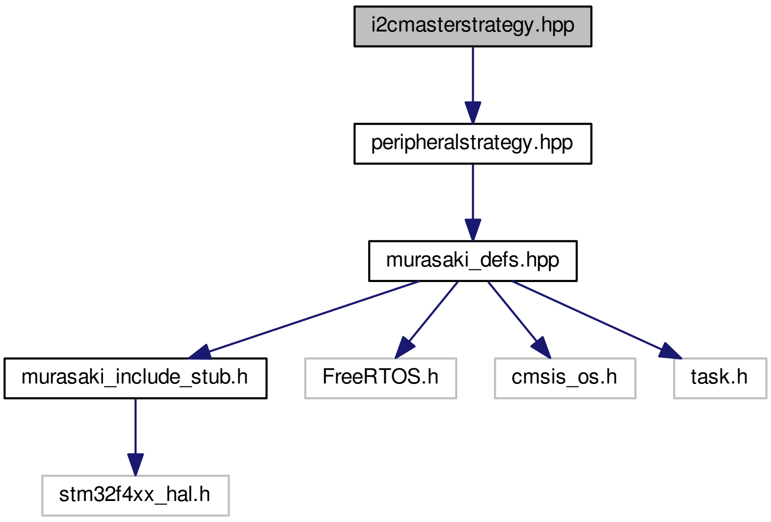
Author

: Seiichi "Suikan" Horie

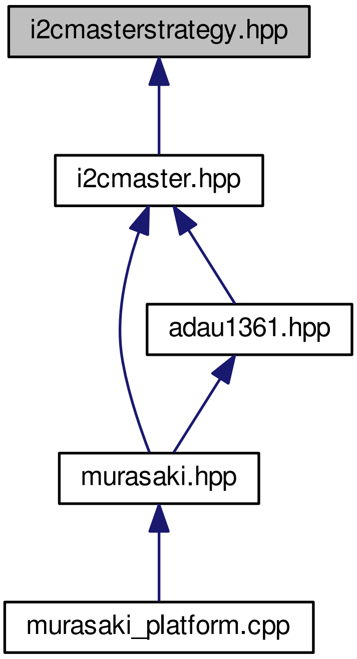
8.19 i2cmasterstrategy.hpp File Reference

Root class definition of the I2C Master.

`#include <peripheralstrategy.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.

8.19.1 Detailed Description

Date

2018/02/11

Author

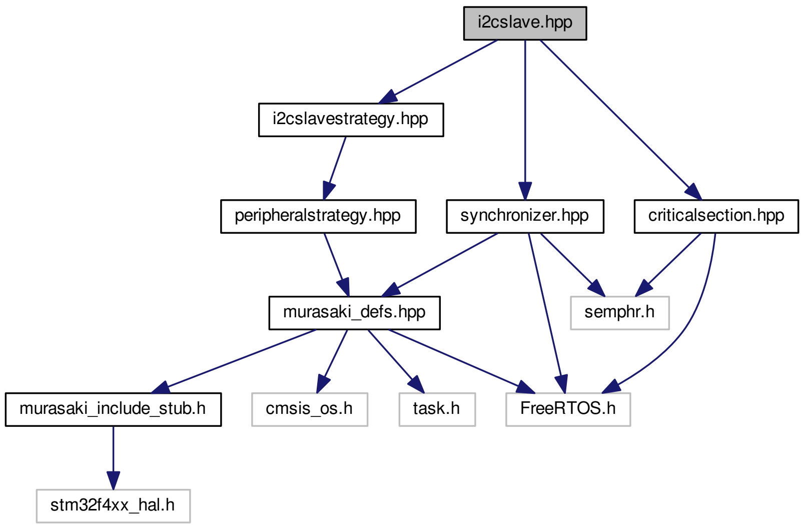
: Seiichi "Suikan" Horie

8.20 i2cslave.hpp File Reference

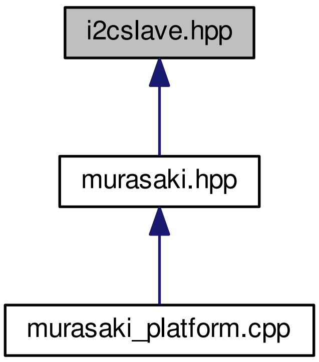
I2C slave. Thread safe, synchronous IO.

```
#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 Free↩ RTOS.

Namespaces

- murasaki

  Personal Platform parts collection.

8.20.1 Detailed Description

Date

2018/10/07

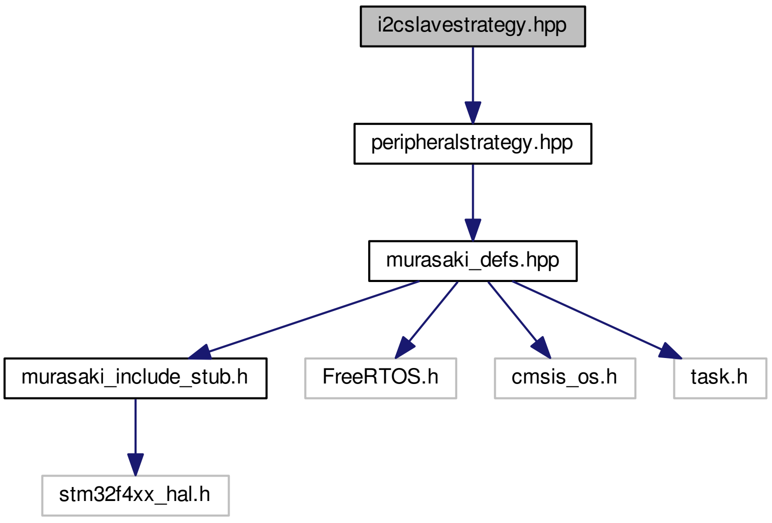
Author

Seiichi "Suikan" Horie

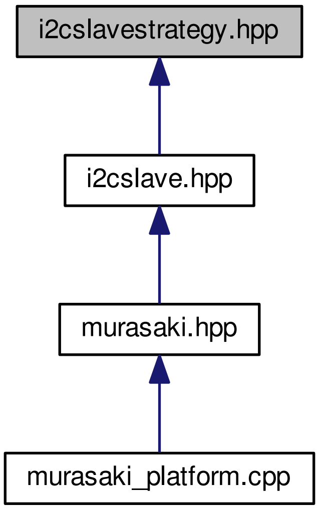
8.21 i2cslavestrategy.hpp File Reference

Root class definition of the I2C Slave.

```
#include <peripheralstrategy.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.

8.21.1 Detailed Description

Date

2018/10/07

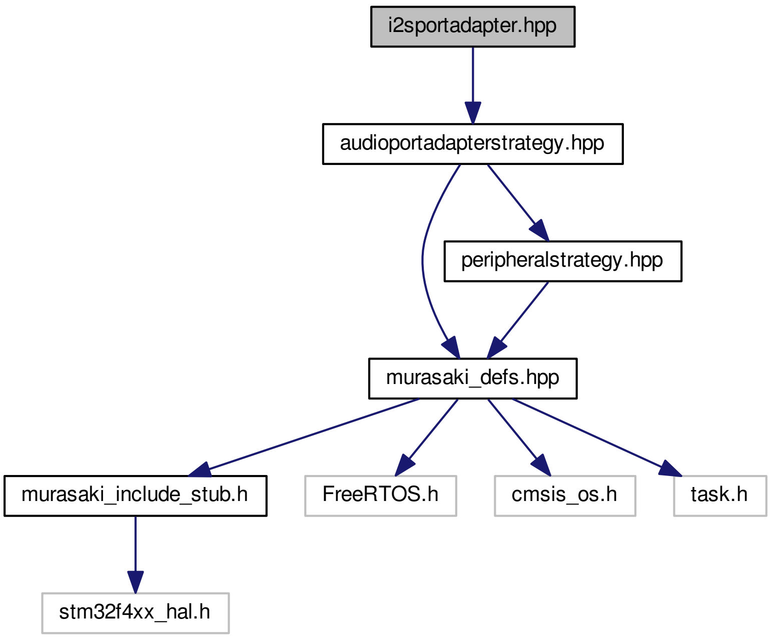
Author

Seiichi "Suikan" Horie

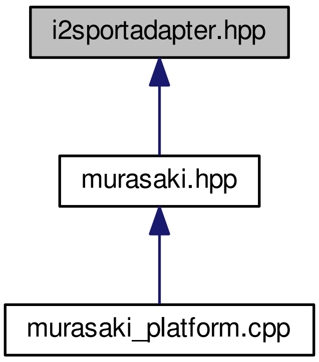
8.22 i2sportadapter.hpp File Reference

```
#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.

8.22.1 Detailed Description

Date

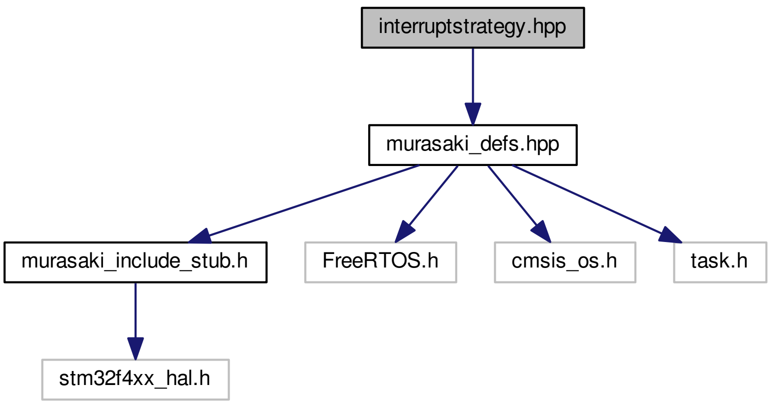
2020/01/15

Author

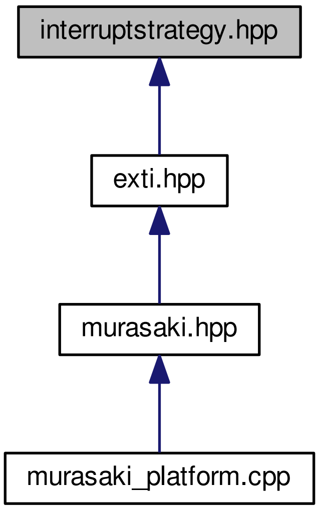
takemasa

8.23 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.

8.23.1 Detailed Description

Date

Feb 29, 2020

Author

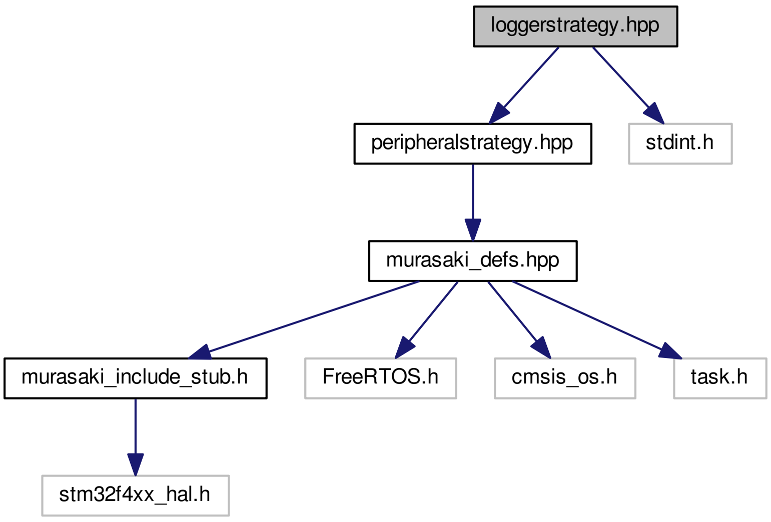
Seiichi "Suikan" Horie

8.24 loggerstrategy.hpp File Reference

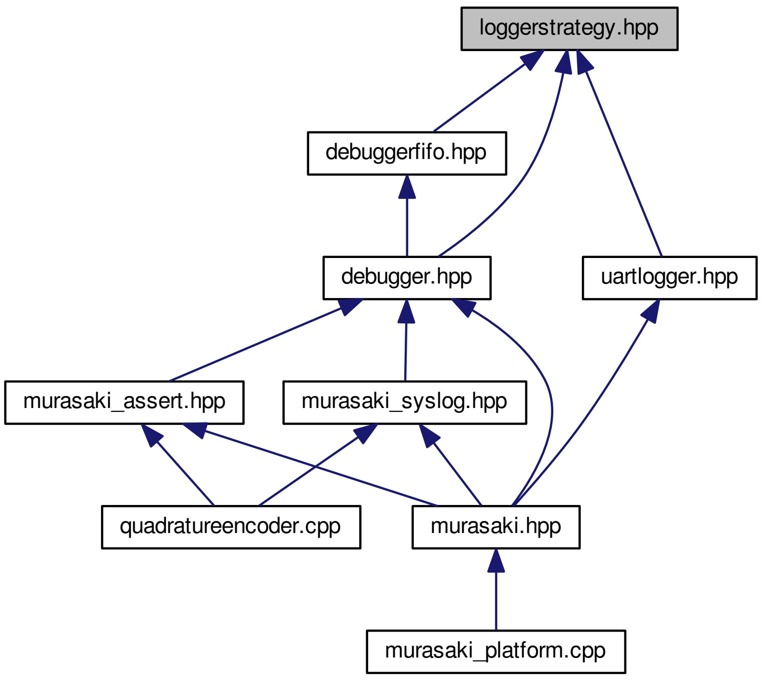
Simplified logging function.

```
#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.

8.24.1 Detailed Description

Date

2018/01/20

Author

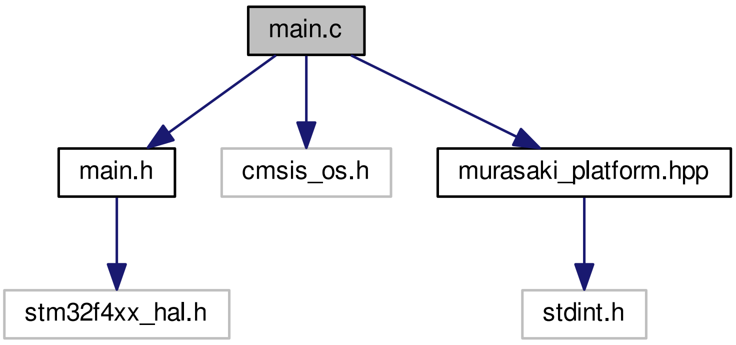
: Seiichi "Suikan" Horie

8.25 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

8.25.1 Detailed Description

Attention

8.25.2 Function Documentation

8.25.2.1 void assert_failed (uint8_t ∗ *file,* uint32_t *line*)

Parameters

file	pointer to the source file name
line	assert_param error line source number

Return values

None	

8.25.2.2 void Error_Handler (void)

Return values

None	

8.25.2.3 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

htim	: TIM handle

Return values

None	

8.25.2.4 int main (void)

Return values

int	

8.25.2.5 void StartDefaultTask (void const * *argument*)

Parameters

argument	Not used

Return values

None	

8.25.2.6 void SystemClock_Config (void)

Return values

None	

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

8.25.3 Variable Documentation

8.25.3.1 DMA_HandleTypeDef hdma_spi1_rx

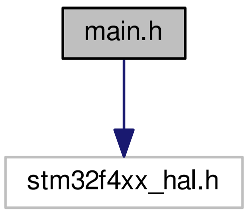
File Name : stm32f4xx_hal_msp.c Description : This file provides code for the MSP Initialization and de-Initialization codes.

Attention

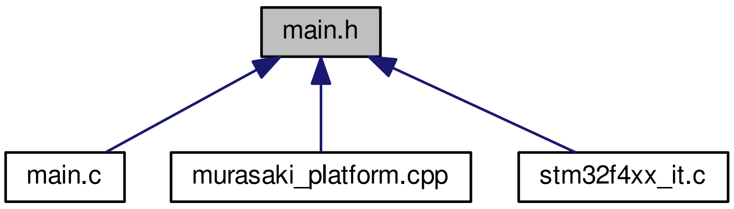
8.26 main.h File Reference

: 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.

8.26.1 Detailed Description

Attention

8.26.2 Function Documentation

8.26.2.1 void Error_Handler (void)

Return values

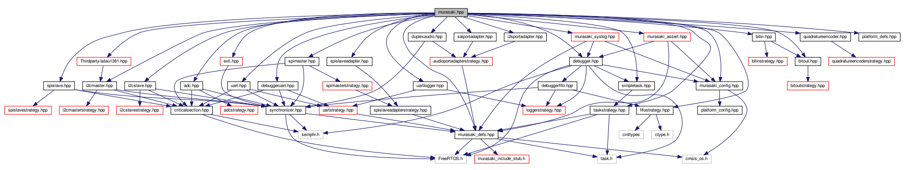
None	

8.27 murasaki.hpp File Reference

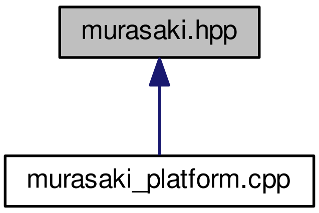
Application include file for Murasaki class library.

```
#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 "saiportadapter.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:



8.27.1 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.

8.28 murasaki_0_intro.hpp File Reference

Doxygen document file. No need to include.

8.28.1 Detailed Description

Date

2020/03/19

Author

Seiichi "Suikan" Horie

8.29 murasaki_1_env.hpp File Reference

Doxygen document file. No need to include.

8.29.1 Detailed Description

Date

2020/03/19

Author

Seiichi "Suikan" Horie

8.30 murasaki_2_ug.hpp File Reference

Doxygen document file. No need to include.

8.30.1 Detailed Description

Date

2018/02/01

Author

Seiichi "Suikan" Horie

8.31 murasaki_3_pg.hpp File Reference

Porting Guide.

8.31.1 Detailed Description

Date

May 25, 2018

Author

Seiichi "Suikan" Horie

8.32 murasaki_4_mod.hpp File Reference

Module definition.

8.32.1 Detailed Description

Date

May 25, 2018

Author

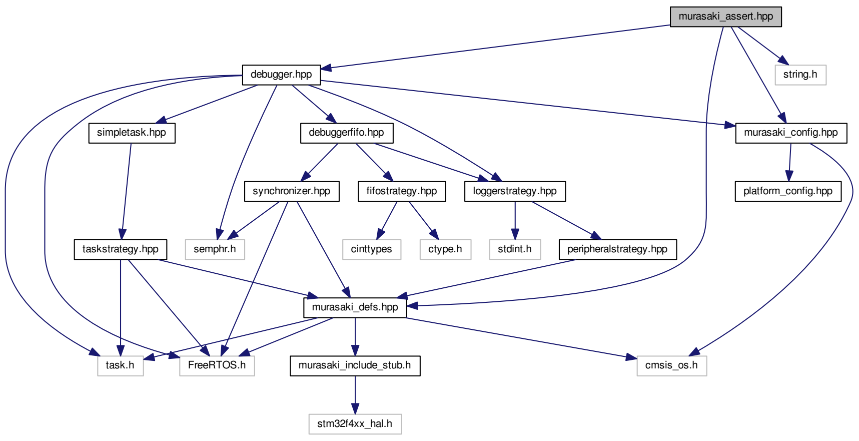
Seiichi "Suikan" Horie

8.33 murasaki_assert.hpp File Reference

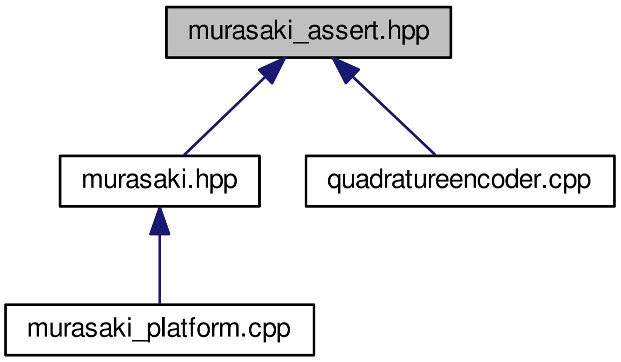
Assertion definition.

```
#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.

8.33.1 Detailed Description

Date

2018/01/31

Author

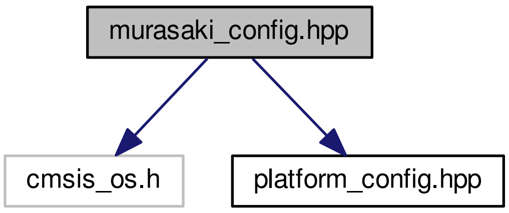
Seiichi "Suikan" Horie

8.34 murasaki_config.hpp File Reference

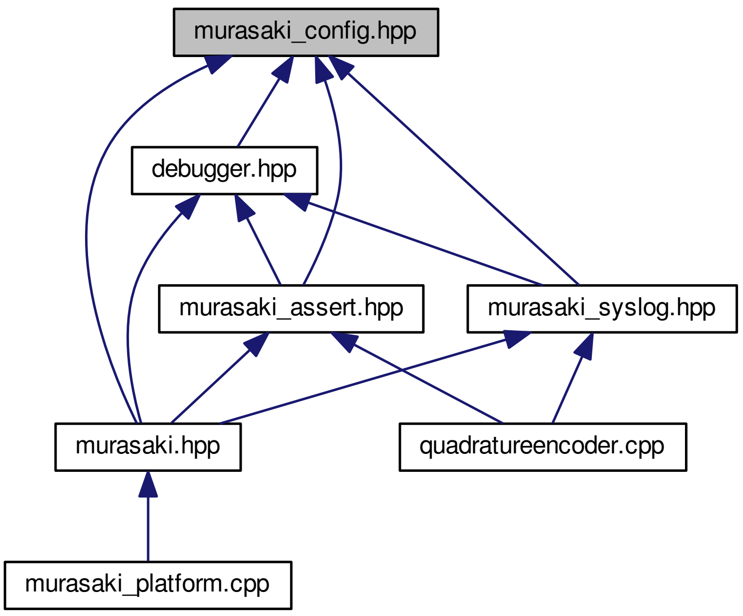
Configuration file for platform.

```
#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 proiority of the debug task.
- #define MURASAKI_CONFIG_NODEBUG false

  Suppress MURASAKI_ASSERT macro.
- #define MURASAKI_CONFIG_NOCYCCNT false

  Doesn't run the CYCCNT counter.

8.34.1 Detailed Description

Date

2018/01/03

Author

Seiichi "Suikan" Horie

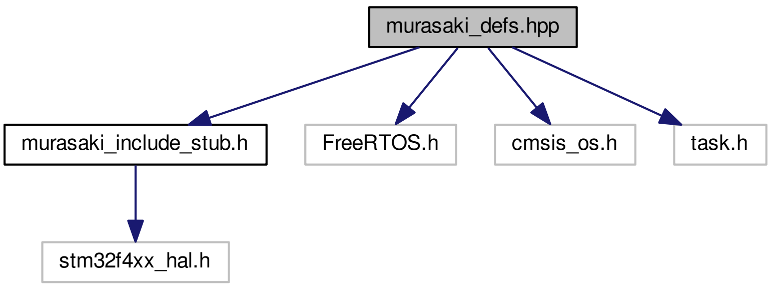
To override the configuration, define the same name macro inside application_config.hpp

8.35 murasaki_defs.hpp File Reference

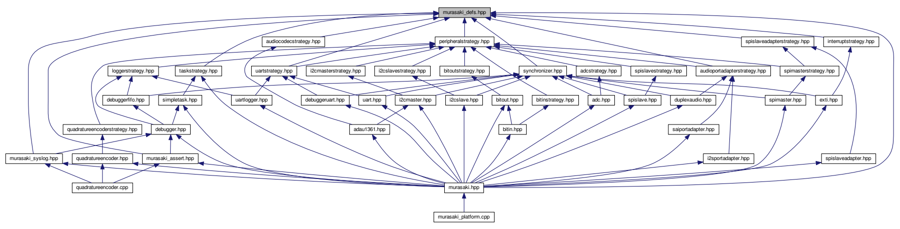
common definition of the platfrom.

```
#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.

8.35.1 Detailed Description

Date

2017/11/05

Author

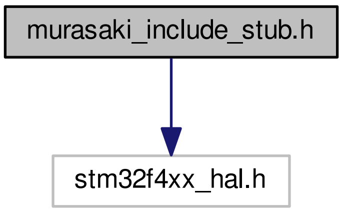
Seiichi "Suikan" Horie

8.36 murasaki_include_stub.h File Reference

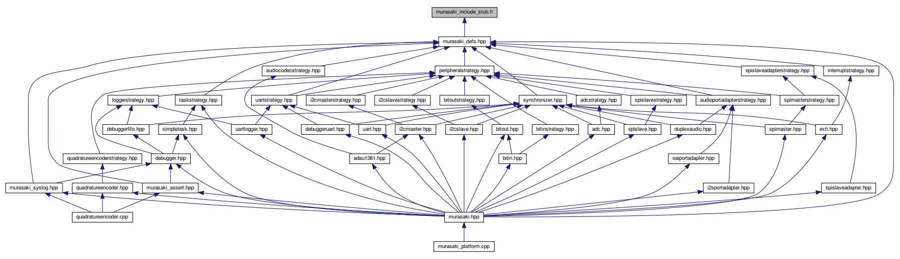
Stub to include the HAL headers.

`#include <stm32f4xx_hal.h>`

Include dependency graph for murasaki_include_stub.h:



This graph shows which files directly or indirectly include this file:



8.36.1 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 appropach is to enumulate 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.

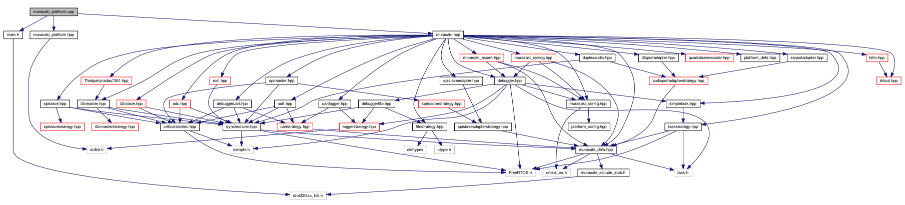
This stub is alternate way. murasaki_defs.hpp is including this file (murasaki_include_stub.h). And this stub file include the appropriate HAL header file. This stub file is generated by murasaki/install script. Thus, user doesn't need to maintain this file.

8.37 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.
- 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 ∗hspi)

  Essential to sync up with SPI.
- void HAL_SPI_ErrorCallback (SPI_HandleTypeDef ∗hspi)

  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_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_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

2018/05/20

Author

Seiichi "Suikan" Horie

8.37.2 Function Documentation

8.37.2.1 void HAL_I2C_MasterRxCpltCallback (I2C_HandleTypeDef ∗ *hi2c*)

Parameters

hi2c	

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.

8.37.2.2 void HAL_I2C_SlaveRxCpltCallback (I2C_HandleTypeDef ∗ *hi2c*)

Parameters

hi2c	

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::ReceiveComplete←Callback() function.

8.37.2.3 void PrintFaultResult (unsigned int ∗ *stack_pointer*)

Parameters

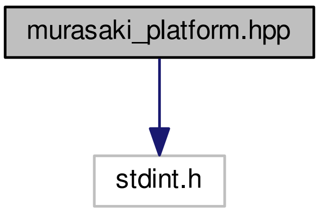
stack_pointer	retrieved stack pointer before interrupt / exception.

Do not call from application. This is murasaki_internal_only.

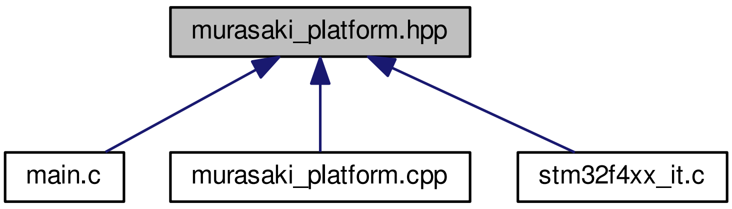
8.38 murasaki_platform.hpp File Reference

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.

8.38.1 Detailed Description

Date

2017/11/12

Author

Seiichi "Suikan" Horie

The resources below are seprented in the murasaki_platform.cpp and serve as glue to the main.c.

8.38.2 Function Documentation

8.38.2.1 void MasterTaskBodyFunction (const void ∗ *ptr*)

Parameters

ptr	Pointer to the parameter block

Task body function of the test. Call test subprogram step by step.

8.38.2.2 void PrintFaultResult (unsigned int ∗ *stack_pointer*)

Parameters

stack_pointer	retrieved stack pointer before interrupt / exception.

Do not call from application. This is murasaki_internal_only.

8.38.2.3 void SlaveTaskBodyFunction (const void ∗ *ptr*)

Parameters

ptr	Pointer to the parameter block

Task body function as demonstration of the murasaki::SimpleTask.

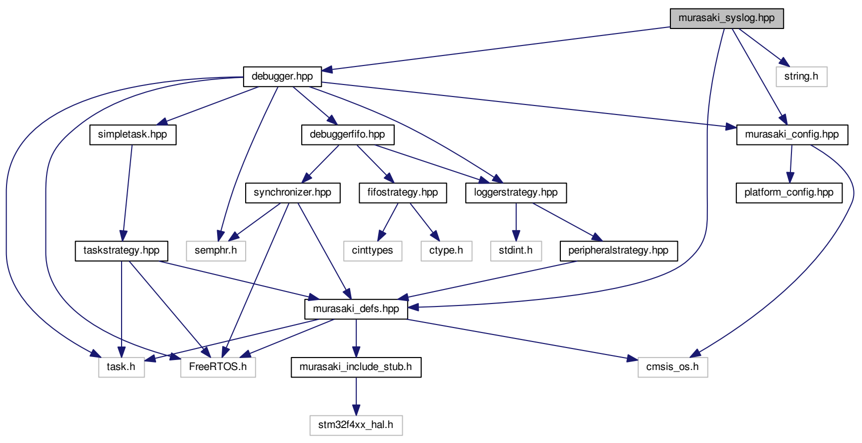
You can delete this function if you don't use.

8.39 murasaki_syslog.hpp File Reference

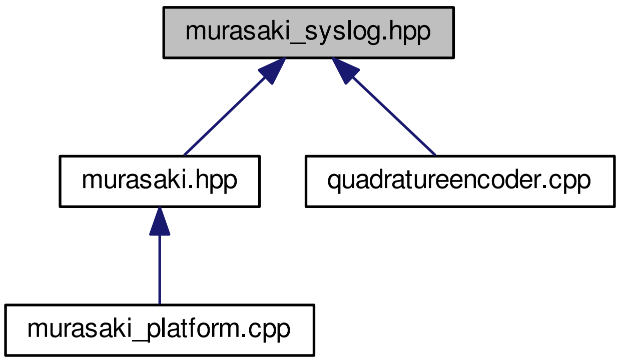
Syslog definition.

```
#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.

8.39.1 Detailed Description

Date

2018/09/01

Author

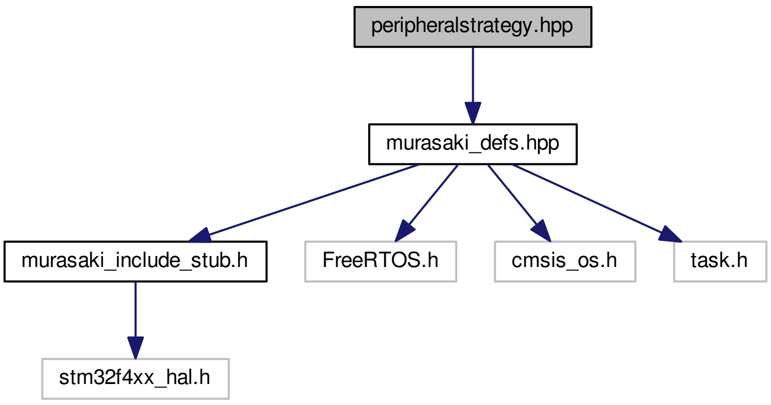
Seiichi "Suikan" Horie

8.40 peripheralstrategy.hpp File Reference

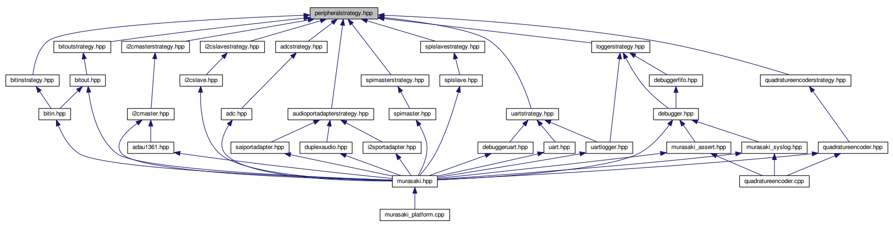
Mother of All peripheral.

```
#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.

8.40.1 Detailed Description

Date

2018/04/26

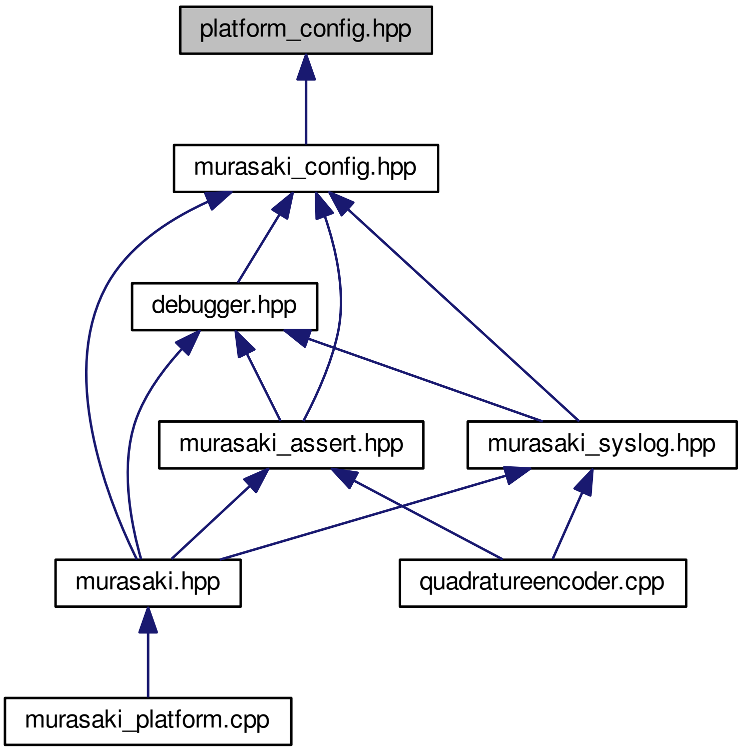
Author

: Seiichi "Suikan" Horie

8.41 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.

8.41.1 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.

8.41.2 Macro Definition Documentation

8.41.2.1 #define MURASAKI_CONFIG_NOSYSLOG false

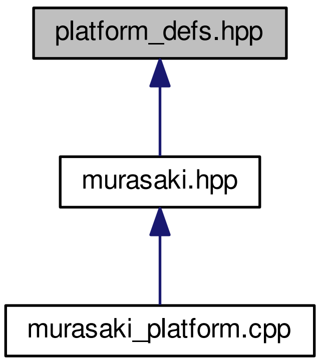
Set this macro to true, to discard the MURASAKI_SYSLOG. Set this macro false, to use the syslog.

To override the definition here, define same macro inside platform_config.hpp.

8.42 platform_defs.hpp File Reference

Murasaki platform customize file.

This graph shows which files directly or indirectly include this file:



Classes

- struct murasaki::Platform

    Custom aggregation struct for user platform.

Namespaces

- murasaki

    Personal Platform parts collection.

Variables

- Platform murasaki::platform

    Grobal variable to provide the access to the platform component.

8.42.1 Detailed Description

Date

2018/01/16

Author

Seiichi "Suikan" Horie

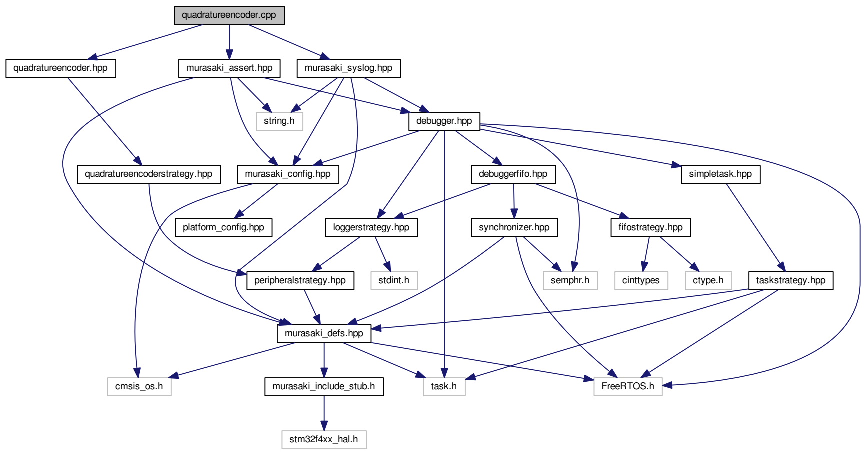
This file contains user defined struct murasaki::Platform.

This file will be included by murasaki.hpp.

8.43 quadratureencoder.cpp File Reference

```
#include "quadratureencoder.hpp"
#include "murasaki_assert.hpp"
#include "murasaki_syslog.hpp"
```

Include dependency graph for quadratureencoder.cpp:



Namespaces

- murasaki

  Personal Platform parts collection.

8.43.1 Detailed Description

Date

2020/02/19

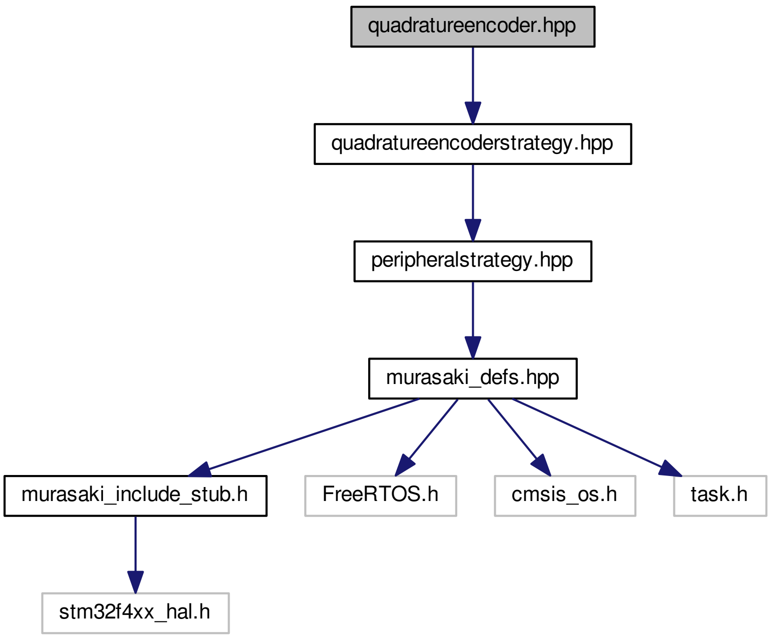
Author

takemasa

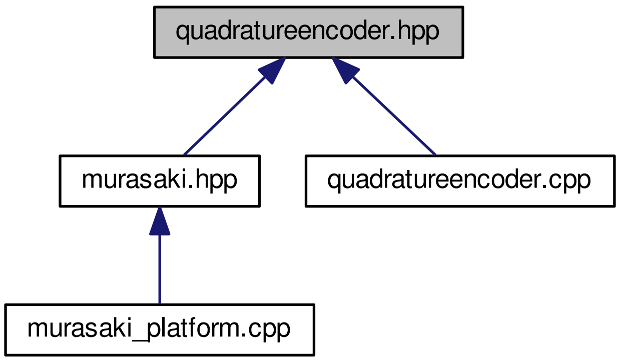
8.44 quadratureencoder.hpp File Reference

```
#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.

8.44.1 Detailed Description

Date

2020/02/19

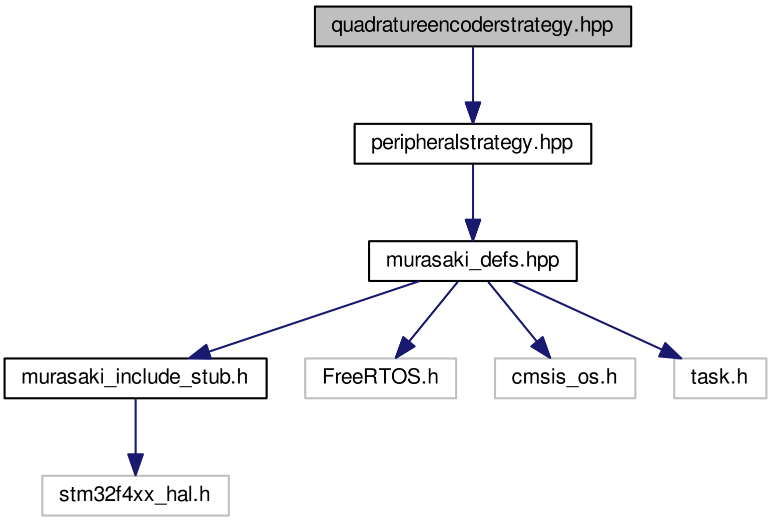
Author

takemasa

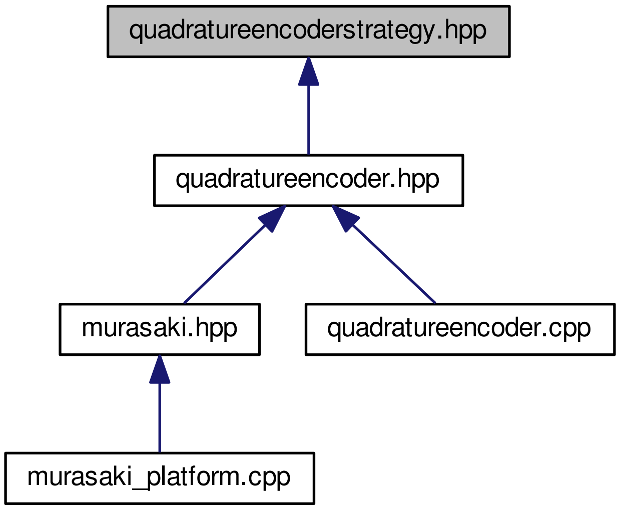
8.45 quadratureencoderstrategy.hpp File Reference

```
#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.

8.45.1 Detailed Description

Date

2020/02/19

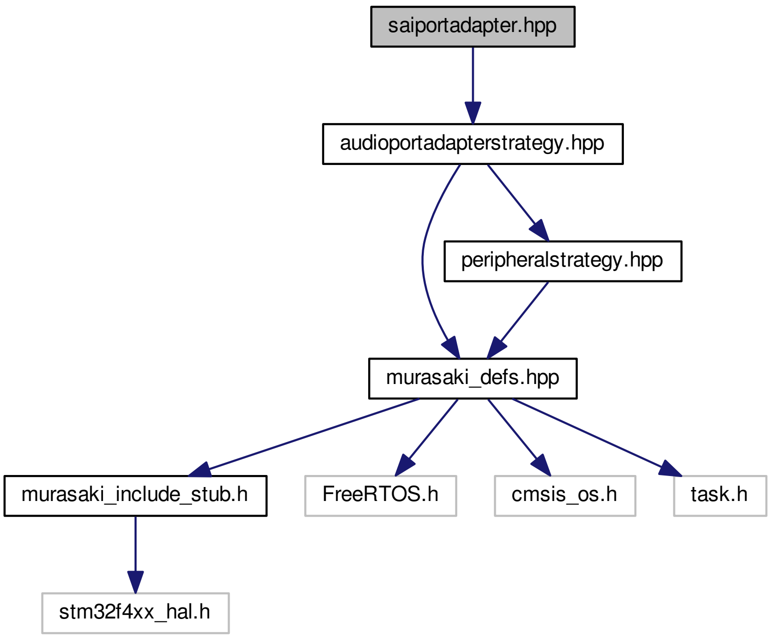
Author

Seiichi Horie

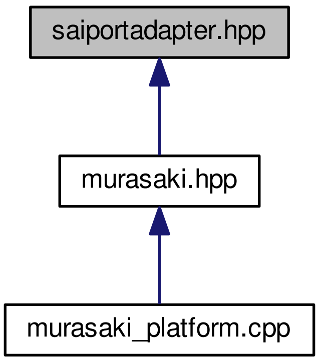
8.46 saiportadapter.hpp File Reference

```
#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.

8.46.1 Detailed Description

Date

2019/07/28

Author

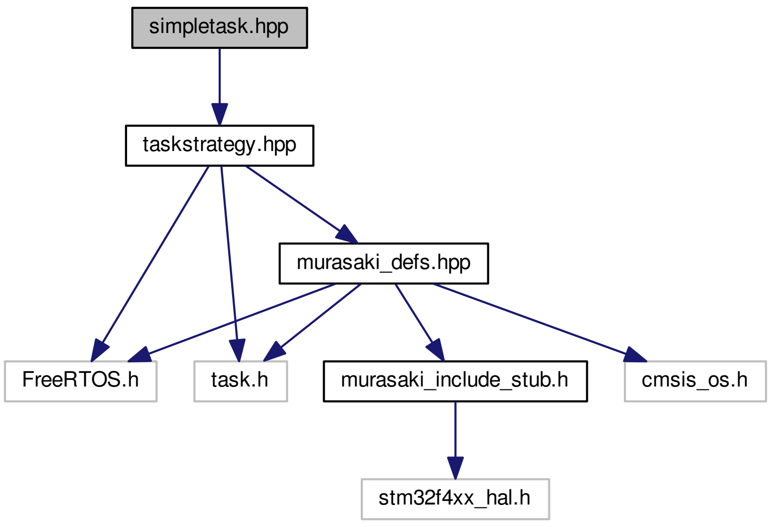
takemasa

8.47 simpletask.hpp File Reference

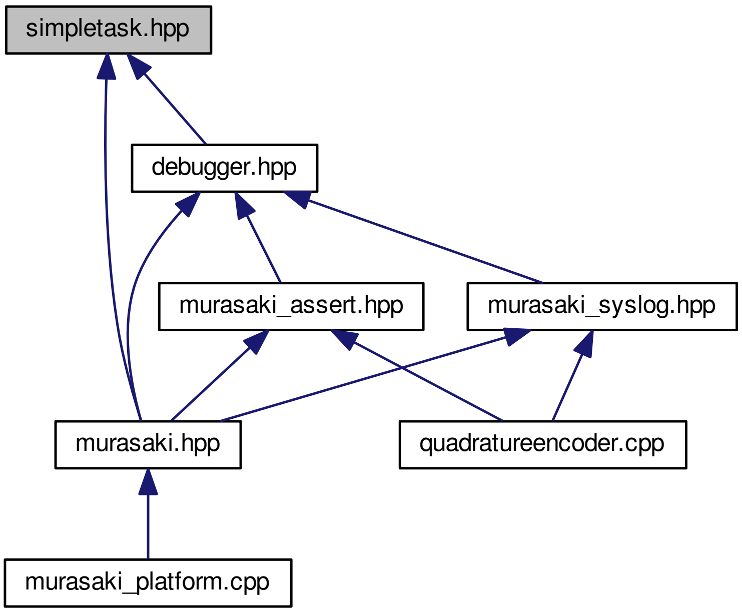
Simplified Task class.

```
#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.

8.47.1 Detailed Description

Date

2019/02/03

Author

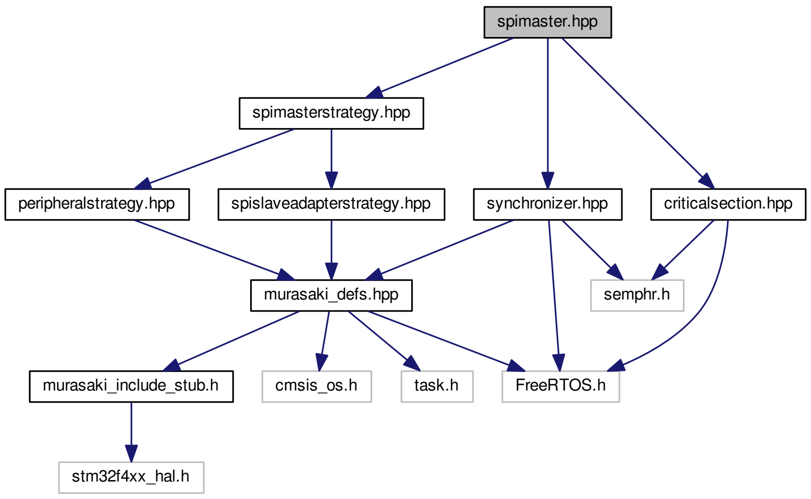
Seiichi "Suikan" Horie

8.48 spimaster.hpp File Reference

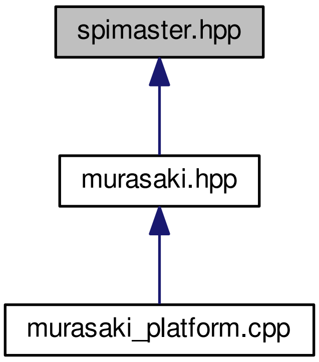
SPI Master. Thread safe and synchronous IO.

```
#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 Free←↩ RTOS.

Namespaces

- murasaki

  Personal Platform parts collection.

8.48.1 Detailed Description

Date

2018/02/14

Author

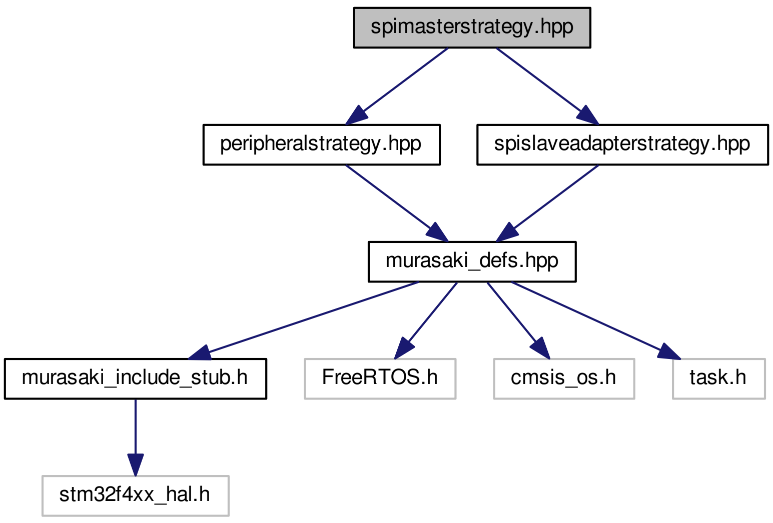
Seiichi "Suikan" Horie

8.49 spimasterstrategy.hpp File Reference

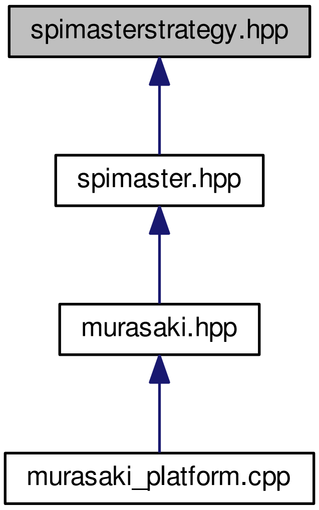
SPI master root class.

```
#include <peripheralstrategy.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.

8.49.1 Detailed Description

Date

2018/02/11

Author

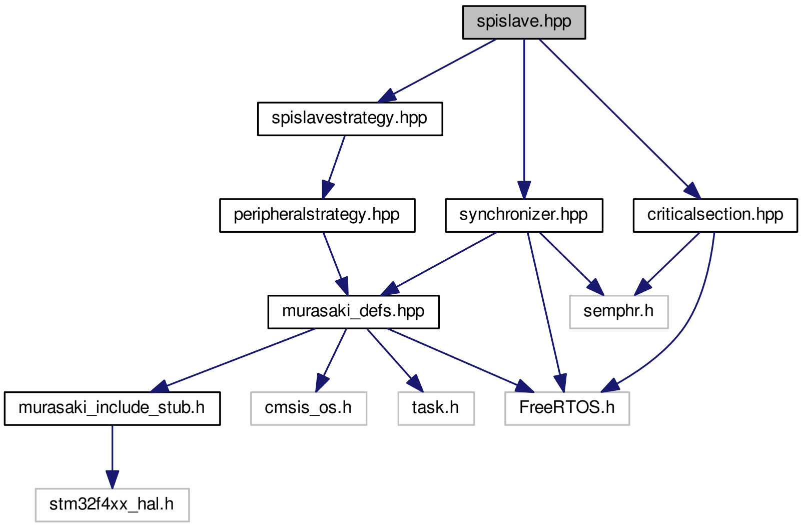
: Seiichi "Suikan" Horie

8.50 spislave.hpp File Reference

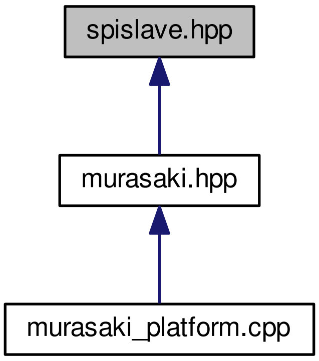
SPI Slave. Thread safe and synchronous IO.

```
#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 Free↩ RTOS.

Namespaces

- murasaki

  Personal Platform parts collection.

8.50.1 Detailed Description

Date

2018/02/14

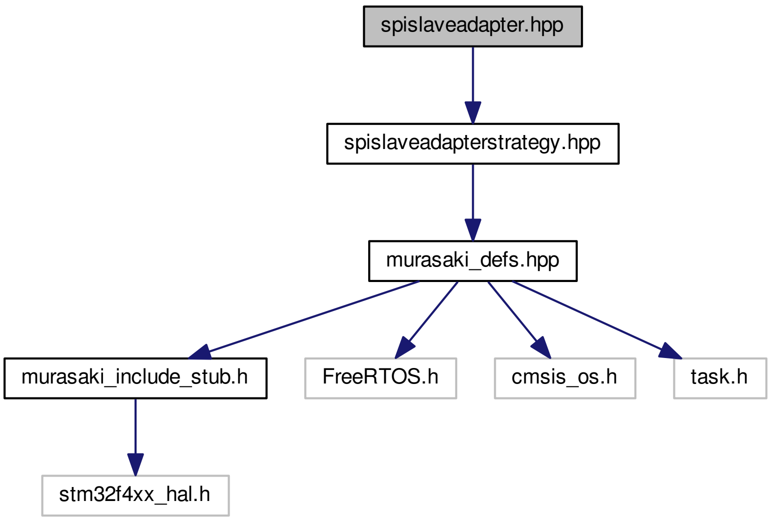
Author

Seiichi "Suikan" Horie

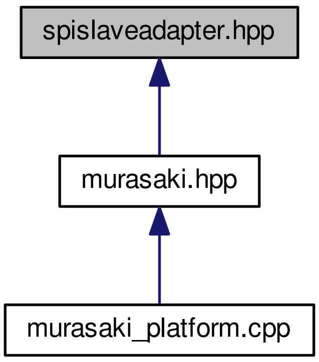
8.51 spislaveadapter.hpp File Reference

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 speficier of SPI slave.

Namespaces

- murasaki

  Personal Platform parts collection.

8.51.1 Detailed Description

Date

2018/02/17

Author

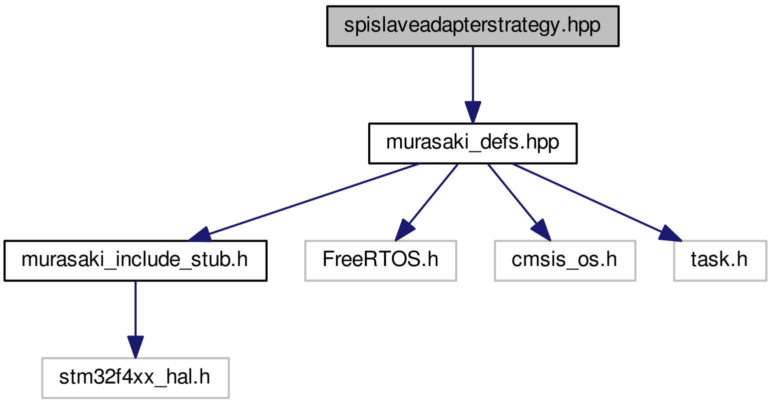
Seiichi "Suikan" Horie

8.52 spislaveadapterstrategy.hpp File Reference

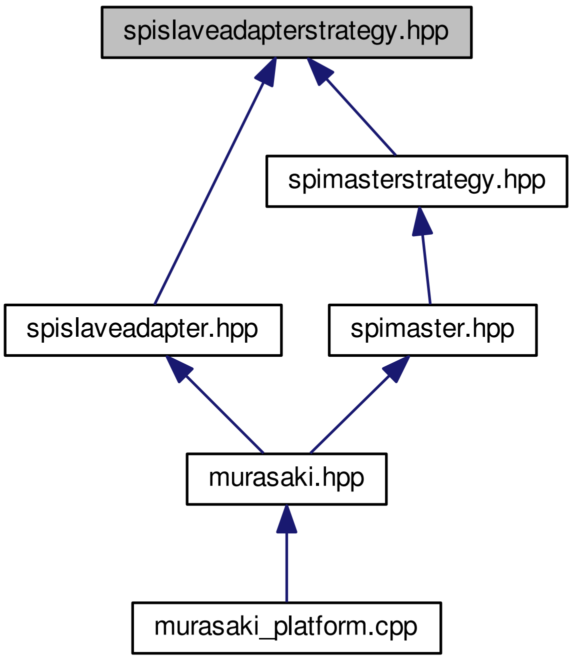
Abstract class of SPI slave specification.

```
#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.

8.52.1 Detailed Description

Date

2018/02/11

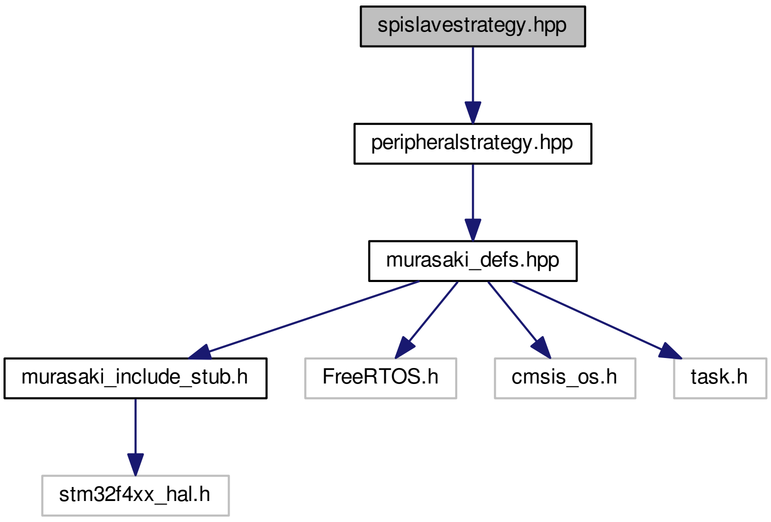
Author

: Seiichi "Suikan" Horie

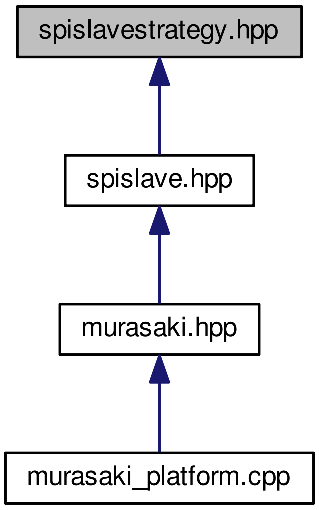
8.53 spislavestrategy.hpp File Reference

SPI master root class.

```
#include <peripheralstrategy.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.

8.53.1 Detailed Description

Date

2018/02/11

Author

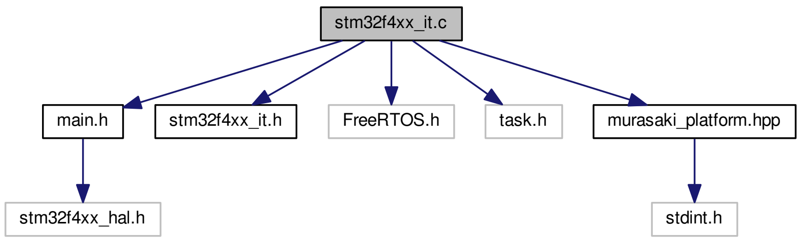
: Seiichi "Suikan" Horie

8.54 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

8.54.1 Detailed Description

Attention

8.54.2 Variable Documentation

8.54.2.1 DMA_HandleTypeDef hdma_spi1_rx

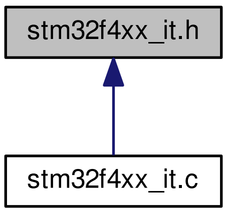
File Name : stm32f4xx_hal_msp.c Description : This file provides code for the MSP Initialization and de-Initialization codes.

Attention

8.55 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.

8.55.1 Detailed Description

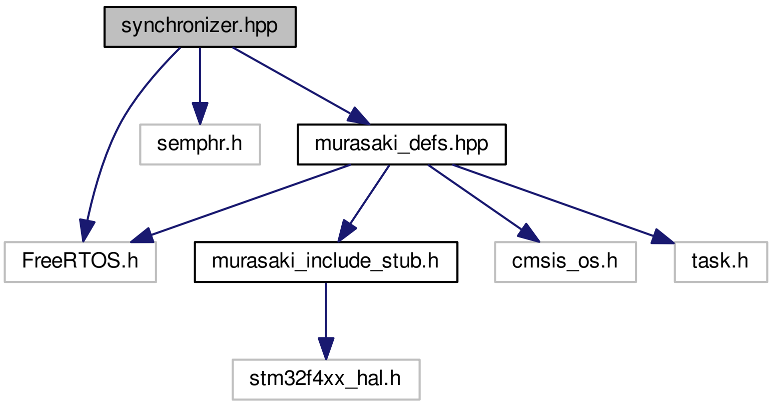
Attention

8.56 synchronizer.hpp File Reference

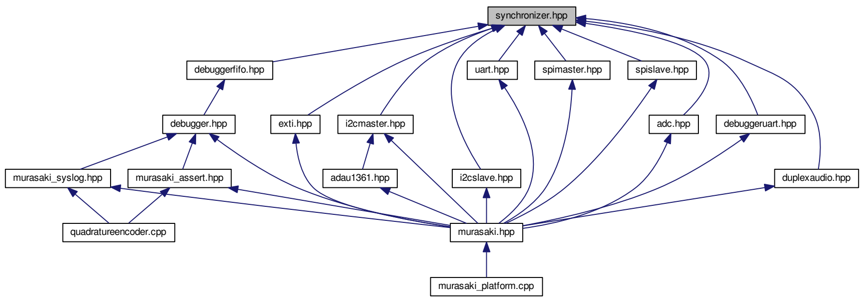
Synchronization between a Task and interrupt.

```
#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.

8.56.1 Detailed Description

Date

2018/01/26

Author

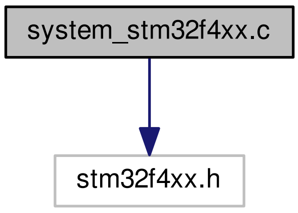
Seiichi "Suikan" Horie

8.57 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.

8.57.1 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.

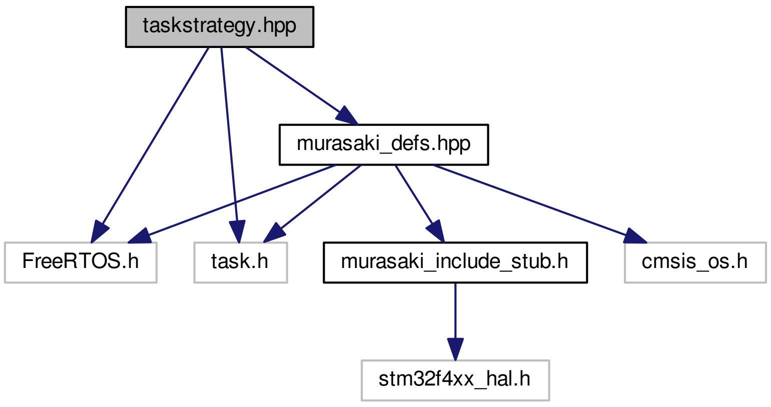
Attention

8.58 taskstrategy.hpp File Reference

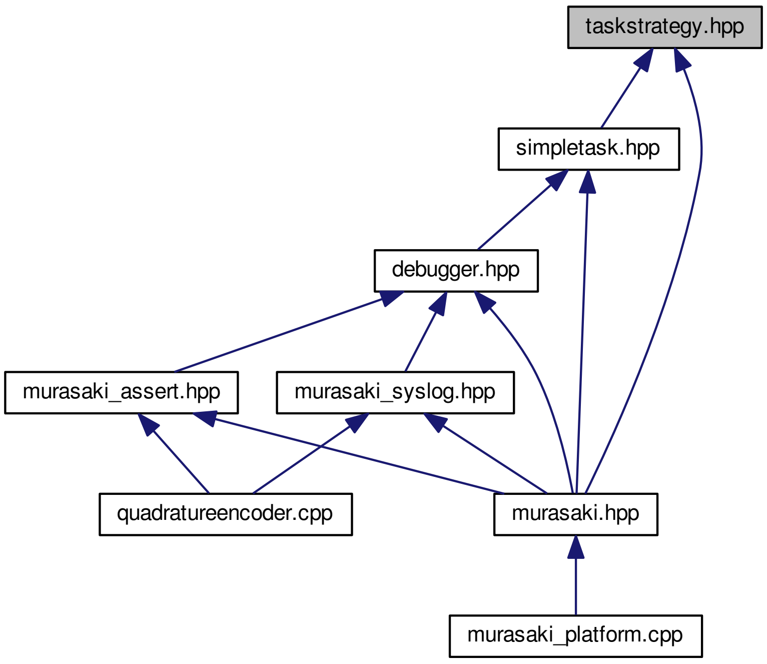
Mother of All Tasks.

```
#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.

8.58.1 Detailed Description

Date

2018/02/20

Author

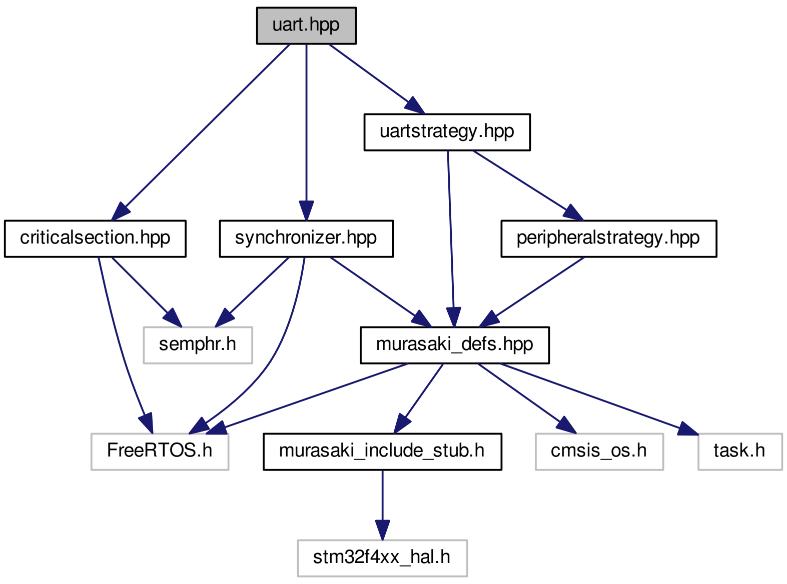
: Seiichi "Suikan" Horie

8.59 uart.hpp File Reference

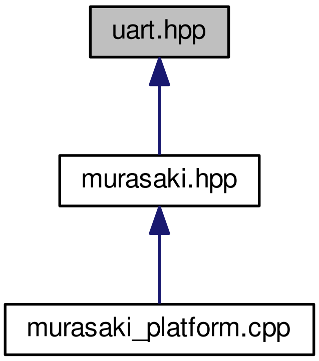
UART. Thread safe and synchronous IO.

```
#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.

8.59.1 Detailed Description

Date

2017/11/05

Author

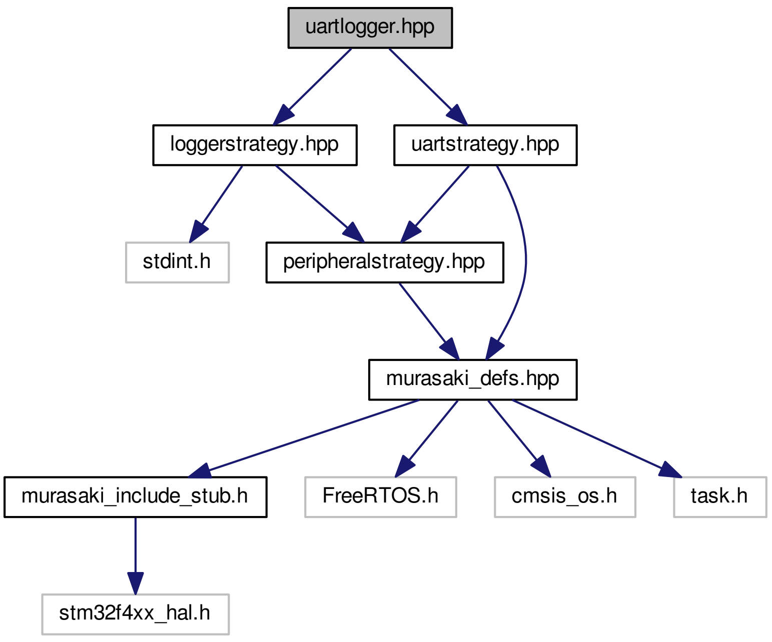
Seiichi "Suikan" Horie

8.60 uartlogger.hpp File Reference

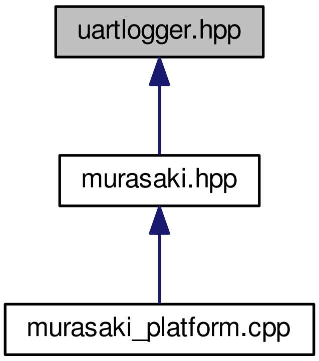
Logging to Uart.

```
#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.

8.60.1 Detailed Description

Date

2018/01/20

Author

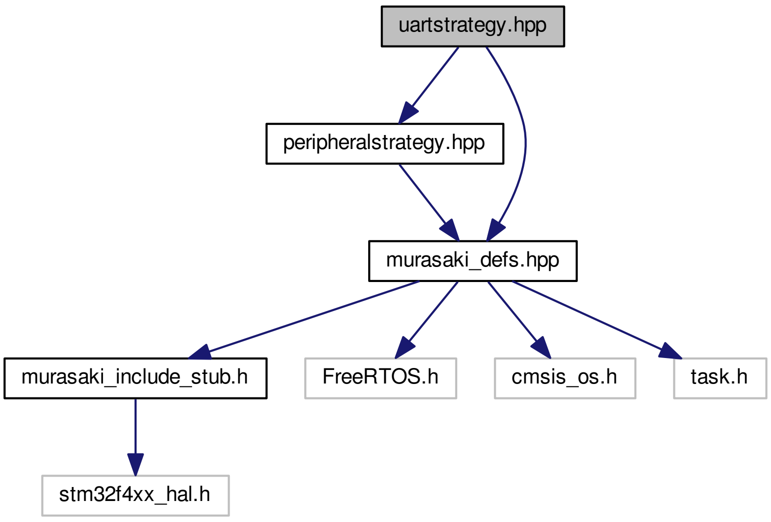
: Seiichi "Suikan" Horie

8.61 uartstrategy.hpp File Reference

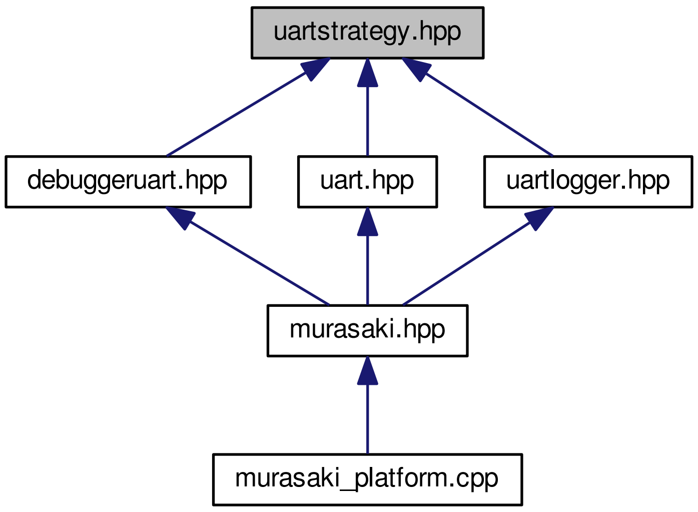
Root class definition of the UART driver.

```
#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.

8.61.1 Detailed Description

Date

2017/11/04

Author

: Seiichi "Suikan" Horie

Index