Murasaki Class Library

1.0.1

Generated by Doxygen 1.8.11

Contents

Chapter 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 IO function is packaged by class types. 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 generated by the CubeIDE.

The STM32Cube HAL is quite rich and flexible. On the other hand, it is quite huge and complex. The classes in Murasaki simplifies 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 easier than using HAL directly.

1.2 Preemptive multi-task

The Murasaki class library is buit on FreeRTOS's preemptive configuration. As a result, Murasaki is automatically aware with 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 completion.

The multi-task programming helps to divide a bigger program to sub-units. This is a good way to develop a large program easier. And the more important point, it is easier to maintain.

1.3 synchronous IO

The synchronous IO is one of the most important 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 completely. 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 true transmission of the last byte. In this case, system generates a completion interrupt while the data is still in FIFO of the peripheral. This is a hardware issue.

To provide the synchronous IO, some member functions are restricted to use only in the task context.

1.4 Thread safe IO

The synchronous IO and the preemptive multi-task provide easier 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, each peripheral wrapping class has exclusive access mechanism by mutex.

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

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 addtion to these two, MURASAKI_SYSLOG macro is avairable.

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 buffer.
- 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 allows easy assertion and logging. This macro uses the murasaki::debugger->Printf() internally.

This assertion is used inside Murasaki class library. 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 application can use this mechanism.

1.8 Configurable

Murasaki is configurable from the two points of view.

First, Musaraki's modules enable only when the relevant peripheral is generated by CubeIDE. This allows you set the CubeIDE to generate only the used peripheral's source code. Such the setting makes total source code smaller. All unused drivers are invisible. For example, if you don't enable the I2C pins on CubeIDE, application programmer cannot see the I2C class in the Murasaki library.

The Second part of the configurable characteristics is Murasaki itself. The programmer can customize the Murasaki for example, task stack size.

Chapter 2

Target and Environment

Murasaki library was originally developed with following environment:

- `Nucleo F722ZE ( STM32F722ZE : Cortex-M7 )`
- `STM32CubeIDE 1.1.0`
- `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+ )`

Chapter 3

Usage Introduction

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

- Message output
- Serial communication
- Debugging with Murasaki.
- Tasking
- Other peripheral
- 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 work with DMA.

These are requirement from the Murasaki library.

3.1 Message output

The Murasaki library has a Printf() like message output mechanism.

This mechanism is 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 formating library. So, programmer can output a message as 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 is a method of murasaki::Debugger class. The murasaki::debugger variable is a one of two Murasaki's golobal variable. And it provide an easy to use message output.

The end-of-line charater is depend on the terminal. In the above sample, the terminator is
. This is for the linux based kermit. Other terminal system may need other 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 debugger message output for an embeded realtime system. Thenk this function is :

- Thread safe
- Asynchronous
- Blocking
- Buffered

In the other word, 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 just passing its address and size.

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

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

Beside of transmit, also Receive() member function exists.

3.3 Debugging with Murasaki.

As we saw, 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. murasaki::debugger->prrntf() has several good feature.

- 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(), edit the stm32fxx_hal_conf.h in the Inc directory. This file is generated by CubeIDE. You can find USE_FULL_ASERT macro as comment out. By declaring this macro, assert_failure is enabled.

```
#define USE_FULL_ASSERT    1
```

And then, you should modify assert_failure() in main.c, to call output function (Note, this modification is altered by the murasaki/install script. While the install script works well, still the USE_FULL_ASSERT macro is a responsibility of the porting programmer).

```
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 the case of the User's Guide, logging is done through the UART port.

Time by time, you may not want to connect a serial terminal to the board unless you have a problem. That means when you find a problem and connect your serial terminal, 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 last tens of messages.

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 encapsulate 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 parameter to pass for the constructor. The first parameter is the name of the task in FreeRTOS. The second one is the task stack size. This size is depend on the task body function. The third one is the priority of the new task. The priority have 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. 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 task object is created, you must call Start() member function to start the task.

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

Then, new task starts.

3.5 Other peripheral

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. Definition and initialization of this member variable are the responsibility of programmer.

In addition to the Transmit(), murasaki::I2cMaster class has Receive(), and 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. Definition and initialization of this member variable are the responsibility of programmer.

In addition to the Transmit(), murasaki::I2cSlave class has 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 flexibility. The SPI master controller must adapt to the several variation of the SPI communication.

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

The flexibility to above configurations need special mechanism. In Murasaki, this flexibility is responsibility of the murasaki::SpiSlaveAdapter class. This class holds these configuration. Then, passed to the 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::TransmitAndRecieve() 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. Definition and initialization of these member variables are the responsibility of 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 "i2c_slave" in the murasaki::platform variable. Definition and initialization of this member variable are the responsibility of programmer.

3.5.5 GPIO

murasaki::BitOut and murasaki::BitIn provides the GPIO functionality

Following is the 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. Definition and initialization of this member variable are the responsibility of programmer.

In addition to the Toggle(), BitIn has Set() and Clear() member function.

3.5.6 Duplex Audio

murasaki::DuplexAudio class provide a realtime audio IO for both TX and RX together.

This class needs a murasaki::AudioPortAdapterStrategy object as interface with hardware.

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

See 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.

The audio processing is recommended to run in a task which has murasaki::ktpRealtime priority. The Transmit←
AndReceive 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 3 program flows. The start point of these flows are always inside CubeIDE generated code. 2 out of 3 flows are for debugging. Only 1 flow have to be understood well by an application programmer.

- Application flow
- HAL Assertion flow
- Spurious Interrupt flow
- Assertion flow
- General Interrupt flow
- EXTI flow

3.6.1 Application flow

The application program flow is the main flow of a Murasaki application.

This program flow starts from the StartDefaultTask() in the Src/main.c. The StartDefaultTas() is a default and first task created by CubeIDE. In the 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 extention is .cpp, the murasaki_platfrom.cpp is compiled by C++ compiler while the main.c is compiled by C compiler. This allows programmer uses C++ language. Thus, the InitPlatform() is the good place to initialize the class based variables.

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 sample, the first half of the InitPlatform() is building a murasaki::debugger variable. Because this variable is utilized for the debugging of the entire application, there is a value to make it at first.

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 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 is depend on the Nucleo board. So, the porting programmer have a responsibility to refer the right UART.

The second half of the InitPlatform() is the creation part of the other peripheral object. This part 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 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 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 provies 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() print the filename and line number with 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 regsiter and exception environment. After printing, the system get into the post-motem mode which responses any key from console and then flush out the contents of printf message FIFO.

Note that the CustomDefaultHandler() works correctly only when the both conditoin is met :

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

The Debug build made unexpected stack frame in the entry code of the HardFaultHandler.

3.6.4 Assertion flow

The assertion flow is similar to the Spurious Interrupt flow.

Once assertion is raised, assertion macro raised Hard Fault execption. The Hard Fault exception handler in the Src/st32****_it.c calles CustomDefaultHandler.

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

3.6.5 General Interrupt flow

As described in the HAL manual, STM32Cube HAL handles all peripheral related interrupt , and then, call corresponding callback function.

These call backs are optional from the view point of the peripheral hardware, but essential hook to sync with software.

Murasaki is using these callback to notify the end of processing, to the peripheral class objects. For example, following is the sample of 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. And calling the ReceiveComplete←Callback() of the UART object in the platform. Note that Murasaki object returns true, if the callback member function parameter matches with its own hardware handle.

Note that forwarding this call back to all the relevant peripheral is a responsibility of the porting programmer. To forward the callback to the multiple objects, you can call like this.

```
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

EXTI flow is very similar to the General Interrupt flow except its timing.

While other peripheral raises interrupt after the peripheral instance are created, EXTI peripheral may raise the interrupt before the platform peripherals are ready.

Then, EXTI call back has 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 USER_Btn_Pin in the above example is generated by CubeIDE, when customer labels "USER_Btn" to some EXTI input pin.

Chapter 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 project setting must refer this directory as the source directory.

4.1.2 Inc directory

This directory contains the include files, the 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, the 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, described in the 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 for programmer.

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 which provide "nice looking" aggregation of the class objects.

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.

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 for programmer.

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 for programmer.

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 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. This management includes 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 :

```
Tab => Pinout & Configuration => Middleware => FreeRTOS => Config Parameters Tab => TOTAL_HEAP_SIZE
```

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.

```
Tab => Project Manager => Code Generator => Linker Settings
```

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 :

```
Tab => Project Manager => Code Generator => Linker Settings
```

4.2.3 Task stack size of the default task

The dealt task has very small stack (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:

```
Tab => Pinout & Configuration => Middleware => FreeRTOS => Config Parameters Tab => MINIMAL_STACK_SIZE
```

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.

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.

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.

4.2.7 I2C peripheral

I2C have to be configured as "I2" mode.

The NVIC interrupt have to be enabled.

To configure as I2C device, the primary slave address have to be configured.

4.2.8 EXTI

The corresponding interrupt have to be enabled by NVIC.

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 allocation to the FreeRTOS's heap. There is some merit of the 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.

First of all, 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 sample, we initialize the uart_console member variable which is murasaki::UartStrategy class. The application programmer control the UART2 over 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 sample 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 documenattion 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_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. In 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 uncomment the USE_FULL_ASSERT macro inside stm32xxxx_hal_conf.h. The file name is depend on the target microprocessor. Thus, the porting programmer have to search the all files inside project.

At the time of 2019/Dec, this definition is in the one for the following files :

- stm32f0xx_hal_conf.h
- stm32f3xx_hal_conf.h
- stm32f4xx_hal_conf.h
- stm32f7xx_hal_conf.h
- stm32g0xx_hal_conf.h
- stm32g4xx_hal_conf.h
- stm32h7xx_hal_conf.h
- stm32l1xx_hal_conf.h
- stm32l4xx_hal_conf.h

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

Chapter 5

Module Index

5.1 Modules

Here is a list of all modules:

Chapter 6

Namespace Index

6.1 Namespace List

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

Chapter 7

Hierarchical Index

7.1 Class Hierarchy

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

Chapter 8

Class Index

8.1 Class List

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

Chapter 9

File Index

9.1 File List

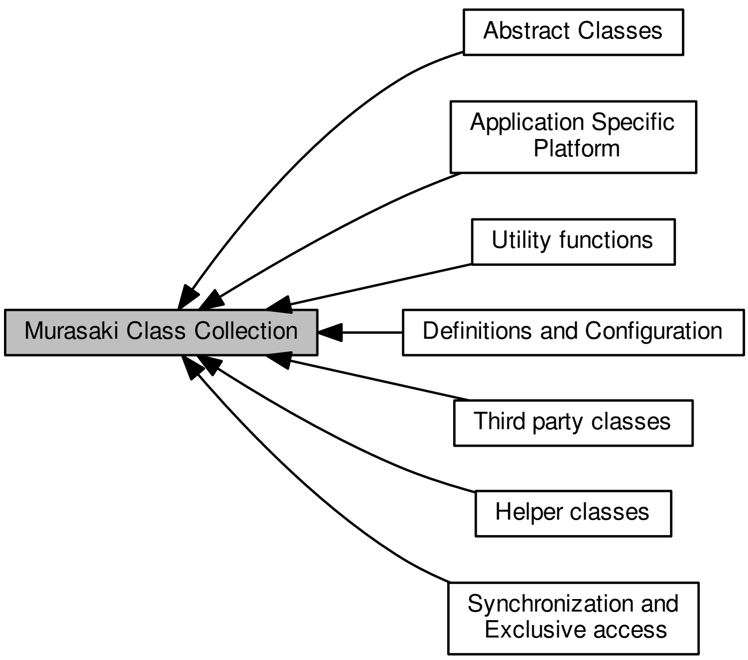
Here is a list of all documented files with brief descriptions:

Chapter 10

Module Documentation

10.1 Murasaki Class Collection

Collaboration diagram for Murasaki Class Collection:



Modules

- Synchronization and Exclusive access
- Third party classes
- Definitions and Configuration
- Application Specific Platform
- Abstract Classes
- Helper classes
- Utility functions

Classes

- class murasaki::BitIn
- struct murasaki::GPIO_type
- class murasaki::BitOut
- class murasaki::Debugger
- class murasaki::DuplexAudio
- class murasaki::I2cMaster
- class murasaki::I2cSlave
- class murasaki::SaiPortAdaptor
- class murasaki::SimpleTask
- class murasaki::SpiMaster
- class murasaki::SpiSlave
- class murasaki::SpiSlaveAdapter
- class murasaki::Uart
- class murasaki::UartLogger

Macros

- #define MURASAKI_ASSERT(COND)
- #define MURASAKI_PRINT_ERROR(ERR)
- #define MURASAKI_SYSLOG(OBJPTR, FACILITY, SEVERITY, FORMAT, ...)

10.1.1 Detailed Description

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

Alternatively, the Usage Introduction is provided to study step by step.

10.1.2 Macro Definition Documentation

10.1.2.1 #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();}\
    }
```

Assert the COND is true.

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.

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

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

This assertion do nothing if programmer defines MURASAKI_CONFIG_NODEBUG macro as true. This macro is defined in the file platform_config.hpp.

10.1.2.2 #define MURASAKI_PRINT_ERROR(*ERR*)

Value:

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

Print ERR if ERR is true.

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 do nothing if programmer defines MURASAKI_CONFIG_NODEBUG macro as true. This macro is defined in the file platform_config.hpp.

For example, following code is typical usage of this macro. ERROR maccro 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
     }
}
```

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

output The debug message

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 thereshold 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 :

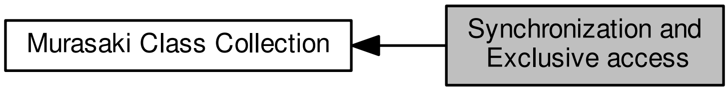
- murasaki::kseDebug for Development/Debug message for tracing normal operation.
- murasaki::kseWarning for relatively severe condition which need abnormal action, or cannot handle.
- murasaki::kseError for falty condtion from HAL or hardware.
- murasaki::kseEmergency for software logic error like assert 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 infromation

10.2 Synchronization and Exclusive access

Collaboration diagram for Synchronization and Exclusive access:



Classes

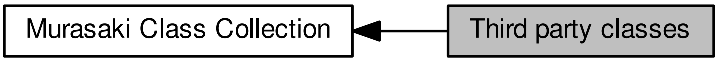
- class murasaki::CriticalSection
- class murasaki::Synchronizer

10.2.1 Detailed Description

These classes are used as parts of the other classes.

10.3 Third party classes

Collaboration diagram for Third party classes:



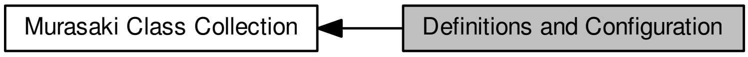
Classes

- class murasaki::Adau1361

10.3.1 Detailed Description

10.4 Definitions and Configuration

Collaboration diagram for Definitions and Configuration:



- #define PLATFORM_CONFIG_DEBUG_LINE_SIZE 256
- #define PLATFORM_CONFIG_DEBUG_BUFFER_SIZE 4096
- #define PLATFORM_CONFIG_DEBUG_SERIAL_TIMEOUT (murasaki::kwmsIndefinitely)
- #define PLATFORM_CONFIG_DEBUG_TASK_STACK_SIZE 256
- #define PLATFORM_CONFIG_DEBUG_TASK_PRIORITY murasaki::ktpHigh
- #define MURASAKI_CONFIG_NODEBUG false
- #define MURASAKI_CONFIG_NOCYCCNT false

10.4.1 Detailed Description

10.4.2 Macro Definition Documentation

10.4.2.1 #define MURASAKI_CONFIG_NOCYCCNT false

Doesn't run the CYCCNT counter.

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.

10.4.2.2 #define MURASAKI_CONFIG_NODEBUG false

Suppress MURASAKI_ASSERT macro.

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.

10.4.2.3 #define PLATFORM_CONFIG_DEBUG_BUFFER_SIZE 4096

Size[byte] of the circular buffer to be transmitted through the serial port.

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.

10.4.2.4 #define PLATFORM_CONFIG_DEBUG_LINE_SIZE 256

Size of one line[byte] in the debug printf.

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.

10.4.2.5 #define PLATFORM_CONFIG_DEBUG_SERIAL_TIMEOUT (murasaki::kwmsIndefinitely)

Timeout of the serial port to transmit the string through the Debug class.

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

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

10.4.2.6 #define PLATFORM_CONFIG_DEBUG_TASK_PRIORITY murasaki::ktpHigh

The task proiority of the debug task.

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.

10.4.2.7 #define PLATFORM_CONFIG_DEBUG_TASK_STACK_SIZE 256

Size[Byte] of the task inside Debug class.

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

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

10.4.3 Enumeration Type Documentation

10.4.3.1 enum murasaki::CodecChannel

Codec channel specifiler.

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

10.4.3.2 enum murasaki::I2cStatus

Return status of the I2C classes.

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

In a single master controler system, you need to care only ki2csNak and ki2csTimeOut. Other error may be caused by multiple master system.

The ki2csNak is returned when one of two happens :

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

The ki2csTimeOUt is returned when slave device streched transfere too long.

The ki2csArbitrationLost is returned when another master won the arbitration. Usulally, the master have to re-try the transfer after certain waiting period.

The ki2csBussError is fatal condition. In the master mode, it could be problem of other device. The root cause is not deterministic. Probably it is 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.

10.4.3.3 enum murasaki::SpiClockPhase

SPI clock configuration for master.

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.

10.4.3.4 enum murasaki::SpiClockPolarity

SPI clock configuration for Master.

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

10.4.3.5 enum murasaki::SpiStatus

Return status of the SPI classes.

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

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

kpisOverflow and the kpisDMA are fatal condition. 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.

10.4.3.6 enum murasaki::SyslogFacility

Category to filter the Syslog output.

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

Internally, these value will be used as bit position in mask.

Enumerator

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
kfaNone Disable all facility.
kfaAll Enable all facility.
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.

10.4.3.7 enum murasaki::SyslogSeverity

Message severity level.

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

10.4.3.8 enum murasaki::TaskPriority

Task class dedicated priority.

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.

10.4.3.9 enum murasaki::UartHardwareFlowControl

Attribute of the UART Hardware Flow Control.

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.

10.4.3.10 enum murasaki::UartStatus

Return status of the UART classes.

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

The Noise error may cuase by the noise on the line.

The overrun may cause when the DMA is too slow or hand shake is not working well.

The DMA error may cause some problem inisde 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.

10.4.3.11 enum murasaki::UartTimeout

This is specific enum for the AbstractUart::Receive() to specify the use of idle line timeout.

The idle line time out is dedicated function of the STM32 peripherals. The interrrupt happens when the receive data is discontinued 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

10.4.3.12 enum murasaki::WaitMilliSeconds : uint32_t

Wait time by milliseconds. For the function which has "wait" or "timeout" parameter.

An uint32_t derived type for specifying wait duration. The integer value represents the waiting duration by miliseconds. Usually a value of this type is passed to some functions as parameter. There are two special cases.

kwmsPolling means function will return immediately regardless of waited event.In other word, with this parameter, function causes time out immediately. Some function may provides the way to know what was the status of the waited event. But some may not.

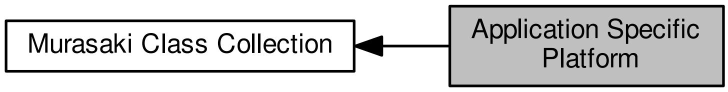
kwmsIndefinitely means function will will not cause time out.

Enumerator

- ***kwmsPolling*** Not waiting. Immediate timeout.
- ***kwmsIndefinitely*** Wait forever.

10.5 Application Specific Platform

Collaboration diagram for Application Specific Platform:



Classes

- struct murasaki::Platform

Functions

- void InitPlatform ()
- void ExecPlatform ()
- void CustomAssertFailed (uint8_t ∗file, uint32_t line)
- void CustomDefaultHandler ()
- 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_SlaveTxCpltCallback (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_GPIO_EXTI_Callback (uint16_t GPIO_Pin)

Variables

- Debugger ∗ murasaki::debugger

10.5.1 Detailed Description

Typical usage of these variables can be seen below. First of all, an .cpp file have 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 to declare 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 murasaki class library. Then, it is mandatory to initialize the debugger member variable.

The following code fragment initialize 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
      );
}
```

10.5.2 Function Documentation

10.5.2.1 void CustomAssertFailed (uint8_t ∗ *file,* uint32_t *line*)

Hook for the assert_failure() in main.c.

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.

10.5.2.2 void CustomDefaultHandler ( )

Hook for the default exception handler. Never return.

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.

10.5.2.3 void ExecPlatform ( )

The body of the real application.

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.

10.5.2.4 void HAL_GPIO_EXTI_Callback (uint16_t *GPIO_Pin*)

Optional interrupt handling of EXTI.

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

10.5.2.5 void HAL_I2C_ErrorCallback (I2C_HandleTypeDef ∗ *hi2c*)

Optional error handling of I2C.

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.

10.5.2.6 void HAL_I2C_MasterTxCpltCallback (I2C_HandleTypeDef ∗ *hi2c*)

Essential to sync up with I2C.

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.

10.5.2.7 void HAL_I2C_SlaveTxCpltCallback (I2C_HandleTypeDef ∗ *hi2c*)

Essential to sync up with I2C.

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.

10.5.2.8 void HAL_SAI_ErrorCallback (SAI_HandleTypeDef ∗ *hsai*)

Optional SAI error interrupt handler.

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.

10.5.2.9 void HAL_SAI_RxCpltCallback (SAI_HandleTypeDef ∗ *hsai*)

Optional SAI interrupt handler at buffer transfer complete.

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.

10.5.2.10 void HAL_SAI_RxHalfCpltCallback (SAI_HandleTypeDef ∗ *hsai*)

Optional SAI interrupt handler at buffer transfer halfway.

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.

10.5.2.11 void HAL_SPI_ErrorCallback (SPI_HandleTypeDef ∗ *hspi*)

Optional error handling of SPI.

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.

10.5.2.12 void HAL_SPI_TxRxCpltCallback (SPI_HandleTypeDef ∗ *hspi*)

Essential to sync up with SPI.

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.

10.5.2.13 void HAL_UART_ErrorCallback (UART_HandleTypeDef ∗ *huart*)

Optional error handling of UART.

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.

10.5.2.14 void HAL_UART_RxCpltCallback (UART_HandleTypeDef * *huart*)

Essential to sync up with UART.

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.

10.5.2.15 void HAL_UART_TxCpltCallback (UART_HandleTypeDef * *huart*)

Essential to sync up with UART.

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.

10.5.2.16 void InitPlatform ( )

Initialize the platform variables.

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.

10.5.3 Variable Documentation

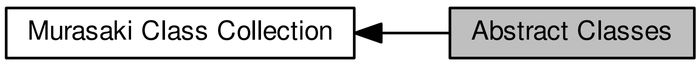
10.5.3.1 murasaki::Debugger * murasaki::debugger

Grobal variable to provide the debugging function.

This variable is declared by murasaki platform. But not instantiated. To make it happen, programmer have to make an variable and initialize it explicitly. Otherwise, Certain debug utility/macro may cause link error, because murasaki::debugger is refered by these utility/macros.

10.6 Abstract Classes

Collaboration diagram for Abstract Classes:



Classes

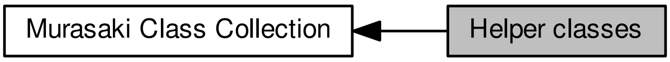
- class murasaki::AudioCodecStrategy
- class murasaki::AudioPortAdapterStrategy
- class murasaki::BitInStrategy
- class murasaki::BitOutStrategy
- class murasaki::FifoStrategy
- class murasaki::I2CMasterStrategy
- class murasaki::I2cSlaveStrategy
- class murasaki::LoggerStrategy
- class murasaki::PeripheralStrategy
- class murasaki::SpiMasterStrategy
- class murasaki::SpiSlaveAdapterStrategy
- class murasaki::SpiSlaveStrategy
- class murasaki::TaskStrategy
- class murasaki::UartStrategy

10.6.1 Detailed Description

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

10.7 Helper classes

Collaboration diagram for Helper classes:



Classes

- class murasaki::DebuggerFifo
- struct murasaki::LoggingHelpers
- class murasaki::DebuggerUart

Functions

- void ∗ operator new (std::size_t size)
- void ∗ operator new[ ] (std::size_t size)
- void operator delete (void ∗ptr)
- void operator delete[ ] (void ∗ptr)

10.7.1 Detailed Description

These classes are not used by customer.

10.7.2 Function Documentation

10.7.2.1 void operator delete (void ∗ *ptr*)

Deallocate the given memory.

Parameters

ptr	Pointer to the memory to deallocate

Returns

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

10.7.2.2 void operator delete[ ] (void * *ptr*)

Deallocate the given memory.

Parameters

ptr	Pointer to the memory to deallocate

Returns

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

10.7.2.3 void* operator new (std::size_t *size*)

Allocate a memory piece with given size.

Parameters

size	Size of the memory to allocate [byte]

Returns

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

10.7.2.4 void* operator new[ ] (std::size_t *size*)

Allocate a memory piece with given size.

Parameters

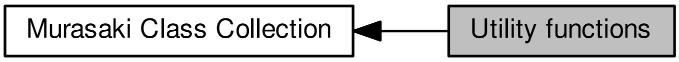
size	Size of the memory to allocate [byte]

Returns

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

10.8 Utility functions

Collaboration diagram for Utility functions:



- static bool murasaki::IsTaskContext ()
- static void murasaki::CleanAndInvalidateDataCacheByAddress (void ∗address, size_t size)
- static void murasaki::CleanDataCacheByAddress (void ∗address, size_t size)
- void murasaki::InitCycleCounter ()
- unsigned int murasaki::GetCycleCounter ()
- static void murasaki::Sleep (unsigned int duration_ms)

10.8.1 Detailed Description

10.8.2 Function Documentation

10.8.2.1 static void murasaki::CleanAndInvalidateDataCacheByAddress (void ∗ *address*, size_t *size*) [inline], [static]

Clean and Flush the specific region of data cache.

Parameters

address	Start address of region
size	Size of region

Keep coherence between the L2 memory and d-cache, between specific region.

The region is specified by address and size. If address is not 32byte aligned, it is truncated to the 32byte alignment, and size is adjusted to follow this alignment.

Once this function is returned, the specific region is coherent.

10.8.2.2 static void murasaki::CleanDataCacheByAddress (void ∗ *address*, size_t *size*) [inline], [static]

Clean the specific region of data cache.

Parameters

address	Start address of region
size	Size of region

Keep coherence between the L2 memory and d-cache, between specific region.

The region is specified by address and size. If address is not 32byte aligned, it is truncated to the 32byte alignment, and size is adjusted to follow this alignment.

Once this function is returned, the specific region is coherent.

10.8.2.3 unsigned int murasaki::GetCycleCounter ( )

Obtain the current cycle count of CYCCNT register.

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.

10.8.2.4 void murasaki::InitCycleCounter ( )

Initialize and start the cycle counter.

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.

10.8.2.5 static bool murasaki::IsTaskContext ( ) [inline], [static]

determine task or ISR context

Returns

true if task context, false if ISR context.

10.8.2.6 static void murasaki::Sleep (unsigned int *duration_ms*) [inline], [static]

Keep task sleeping during the specific duration.

Parameters

duration_ms	Sleeping time by milliseconds.

Whenever this function is called, that task gets into the sleeping (or waiting, the name is up to RTOS) immediately.

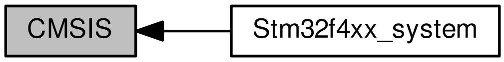
Then, wake up after specified duration.

Note that the duration is interpreted as "at least". The actual sleeping duration could be longer than the specified duration by parameter. The worst error between the actual duration and the specified duration is the period of the tick in system.

For example, if the tick period is 10mS, the worst error is 10mS.

10.9 CMSIS

Collaboration diagram for CMSIS:



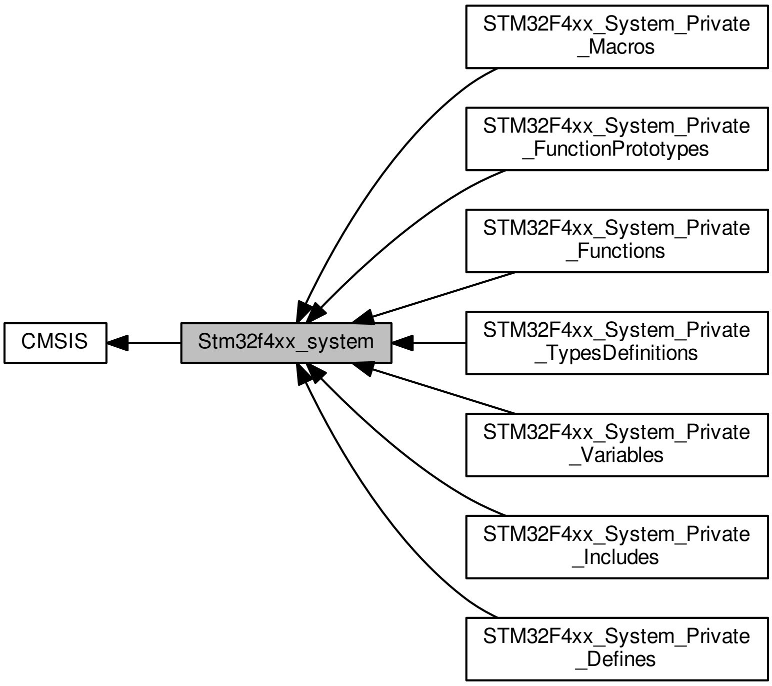
Modules

- Stm32f4xx_system

10.9.1 Detailed Description

10.10 Stm32f4xx_system

Collaboration diagram for Stm32f4xx_system:



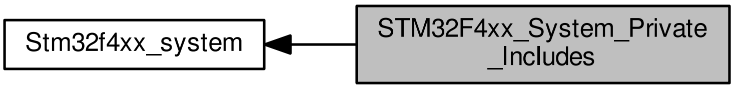
Modules

- STM32F4xx_System_Private_Includes
- STM32F4xx_System_Private_TypesDefinitions
- STM32F4xx_System_Private_Defines
- STM32F4xx_System_Private_Macros
- STM32F4xx_System_Private_Variables
- STM32F4xx_System_Private_FunctionPrototypes
- STM32F4xx_System_Private_Functions

10.10.1 Detailed Description

10.11 STM32F4xx_System_Private_Includes

Collaboration diagram for STM32F4xx_System_Private_Includes:



Macros

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

10.11.1 Detailed Description

10.11.2 Macro Definition Documentation

10.11.2.1 #define HSE_VALUE ((uint32_t)25000000)

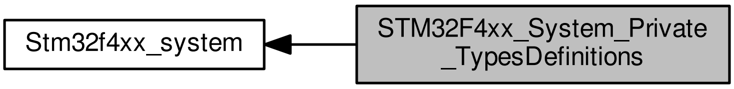
Default value of the External oscillator in Hz

10.11.2.2 #define HSI_VALUE ((uint32_t)16000000)

Value of the Internal oscillator in Hz

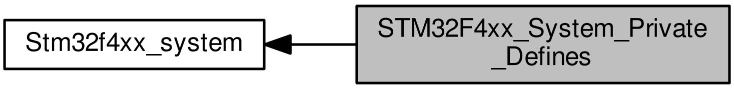
10.12 STM32F4xx_System_Private_TypesDefinitions

Collaboration diagram for STM32F4xx_System_Private_TypesDefinitions:



10.13 STM32F4xx_System_Private_Defines

Collaboration diagram for STM32F4xx_System_Private_Defines:



Macros

- #define VECT_TAB_OFFSET 0x00

10.13.1 Detailed Description

10.13.2 Macro Definition Documentation

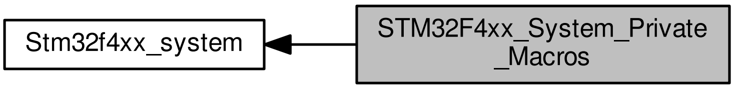
10.13.2.1 #define VECT_TAB_OFFSET 0x00

< 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.

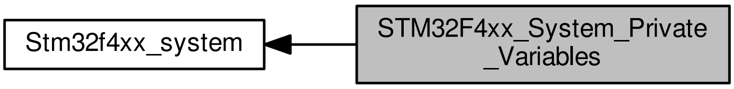
10.14 STM32F4xx_System_Private_Macros

Collaboration diagram for STM32F4xx_System_Private_Macros:



10.15 STM32F4xx_System_Private_Variables

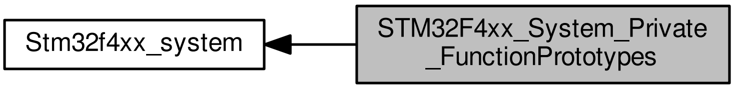
Collaboration diagram for STM32F4xx_System_Private_Variables:



10.15.1 Detailed Description

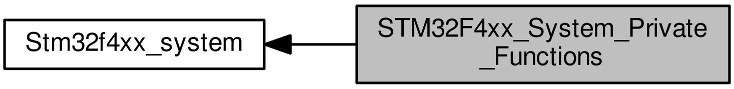
10.16 STM32F4xx_System_Private_FunctionPrototypes

Collaboration diagram for STM32F4xx_System_Private_FunctionPrototypes:



10.17 STM32F4xx_System_Private_Functions

Collaboration diagram for STM32F4xx_System_Private_Functions:



Functions

- void SystemInit (void)
- void SystemCoreClockUpdate (void)

10.17.1 Detailed Description

10.17.2 Function Documentation

10.17.2.1 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.

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	

10.17.2.2 void SystemInit (void)

Setup the microcontroller system Initialize the FPU setting, vector table location and External memory configuration.

Parameters

None	

Return values

None	

Chapter 11

Namespace Documentation

11.1 murasaki Namespace Reference

Classes

- class Adau1361
- class AudioCodecStrategy
- class AudioPortAdapterStrategy
- class BitIn
- class BitInStrategy
- class BitOut
- class BitOutStrategy
- class CriticalSection
- class Debugger
- class DebuggerFifo
- class DebuggerUart
- class DuplexAudio
- class FifoStrategy
- struct GPIO_type
- class I2cMaster
- class I2CMasterStrategy
- class I2cSlave
- class I2cSlaveStrategy
- class LoggerStrategy
- struct LoggingHelpers
- class PeripheralStrategy
- struct Platform
- class SaiPortAdaptor
- class SimpleTask
- class SpiMaster
- class SpiMasterStrategy
- class SpiSlave
- class SpiSlaveAdapter
- class SpiSlaveAdapterStrategy
- class SpiSlaveStrategy
- class Synchronizer
- class TaskStrategy
- class Uart
- class UartLogger
- class UartStrategy

Enumerations

Functions

- void SetSyslogSererityThreshold (murasaki::SyslogSeverity severity)
- void SetSyslogFacilityMask (uint32_t mask)
- void AddSyslogFacilityToMask (murasaki::SyslogFacility facility)
- void RemoveSyslogFacilityFromMask (murasaki::SyslogFacility facility)
- bool AllowedSyslogOut (murasaki::SyslogFacility facility, murasaki::SyslogSeverity severity)

- static bool IsTaskContext ()
- static void CleanAndInvalidateDataCacheByAddress (void ∗address, size_t size)
- static void CleanDataCacheByAddress (void ∗address, size_t size)
- void InitCycleCounter ()
- unsigned int GetCycleCounter ()
- static void Sleep (unsigned int duration_ms)

Variables

- Debugger ∗ debugger

- Platform platform

11.1.1 Detailed Description

This name space 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.

11.1.2 Function Documentation

11.1.2.1 void murasaki::AddSyslogFacilityToMask (murasaki::SyslogFacility *facility*)

Add Syslog facility to the filter mask.

Parameters

facility	Allow this facility to output

See AllowedSyslogOut to understand when the message is out.

11.1.2.2 bool murasaki::AllowedSyslogOut (murasaki::SyslogFacility *facility,* murasaki::SyslogSeverity *severity*)

Check if given facility and severity message is allowed to output.

Parameters

facility	Message facility
severity	Message seveirty

Returns

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

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

If seveirty is higher than or equal to kseError, message is allowed to out.

If the severity is lower than kseError, the message is allowered to out only whhen :

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

11.1.2.3 void murasaki::RemoveSyslogFacilityFromMask (murasaki::SyslogFacility *facility*)

Remove Syslog facility to the filter mask.

Parameters

facility	Deny this facility to output

See AllowedSyslogOut to understand when the message is out.

11.1.2.4 void murasaki::SetSyslogFacilityMask (uint32_t *mask*)

Set the syslog facility mask.

Parameters

mask	Facility bit mask. "1" allows output of the corresponding facility

The parameter is not the facility. A bit mask. By default, the bit mask is 0xFFFFFFFF which allows all facility.

See AllowedSyslogOut to understand when the message is out.

11.1.2.5 void murasaki::SetSyslogSererityThreshold (murasaki::SyslogSeverity *severity*)

Set the syslog severity threshold.

Parameters

severity	

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

11.1.3 Variable Documentation

11.1.3.1 murasaki::Platform murasaki::platform

Grobal variable to provide the access to the platform component.

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.

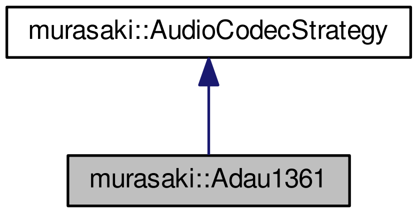
Chapter 12

Class Documentation

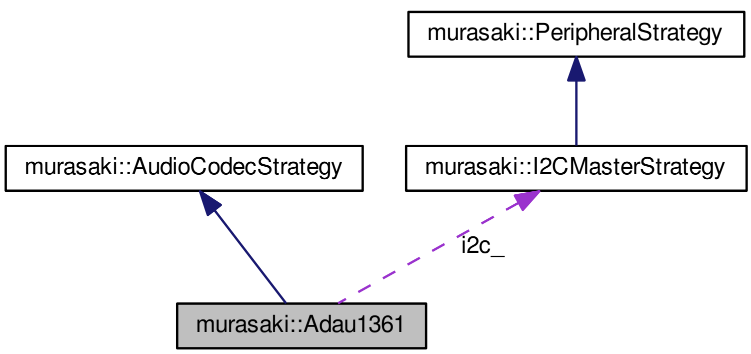
12.1 murasaki::Adau1361 Class Reference

`#include <adau1361.hpp>`

Inheritance diagram for murasaki::Adau1361:



Collaboration diagram for murasaki::Adau1361:



Public Member Functions

- Adau1361 (unsigned int fs, unsigned int master_clock, murasaki::I2CMasterStrategy *controller, unsigned int i2c_device_addr)
- virtual void Start (void)
- virtual void SetGain (murasaki::CodecChannel channel, float left_gain, float right_gain)
- virtual void Mute (murasaki::CodecChannel channel, bool mute=true)
- virtual void SendCommand (const uint8_t command[ ], int size)

Protected Member Functions

- virtual void WaitPllLock (void)
- virtual void SetLineInputGain (float left_gain, float right_gain, bool mute=false)
- virtual void SetAuxInputGain (float left_gain, float right_gain, bool mute=false)
- virtual void SetLineOutputGain (float left_gain, float right_gain, bool mute=false)
- virtual void SetHpOutputGain (float left_gain, float right_gain, bool mute=false)
- virtual void SendCommandTable (const uint8_t table[ ][3], int rows)

12.1.1 Detailed Description

Initialize the ADAU1361 codec based on the given parameter.

12.1.2 Constructor & Destructor Documentation

12.1.2.1 murasaki::Adau1361::Adau1361 (unsigned int *fs*, unsigned int *master_clock*, murasaki::I2CMasterStrategy * *controller*, unsigned int *i2c_device_addr*)

constructor.

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

12.1.3 Member Function Documentation

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

Mute the specific channel.

Parameters

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

Implements murasaki::AudioCodecStrategy.

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

send one command to ADAU1361.

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.

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

send one command to ADAU1361.

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

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

Set the aux input gain and enable the relevant mixer.

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.

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

Set channel gain.

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.

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

Set the headphone output gain and enable the relevant mixer.

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.

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

Set the line input gain and enable the relevant mixer.

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.

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

Set the line output gain and enable the relevant mixer.

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.

12.1.3.9 virtual void murasaki::Adau1361::Start (void) [virtual]

Set up the ADAU1361 codec, and then, start the codec.

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.

12.1.3.10 virtual void murasaki::Adau1361::WaitPllLock (void) [protected], [virtual]

wait until PLL locks.

Service function for the ADAu1361 board implementer.

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

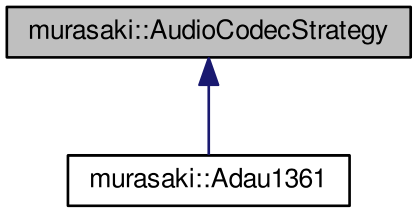
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/Thirdparty/adau1361.hpp

12.2 murasaki::AudioCodecStrategy Class Reference

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

Inheritance diagram for murasaki::AudioCodecStrategy:



Public Member Functions

- AudioCodecStrategy (unsigned int fs)
- virtual void Start (void)=0
- virtual void SetGain (murasaki::CodecChannel channel, float left_gain, float right_gain)=0
- virtual void Mute (murasaki::CodecChannel channel, bool mute=true)=0
- virtual void SendCommand (const uint8_t command[ ], int size)=0

12.2.1 Detailed Description

This class is template for all codec classes

12.2.2 Constructor & Destructor Documentation

12.2.2.1 murasaki::AudioCodecStrategy::AudioCodecStrategy (unsigned int *fs*) `[inline]`

constructor.

Parameters

fs	Sampling frequency.

initialize the internal variables.

12.2.3 Member Function Documentation

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

Mute the specific channel.

Parameters

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

Implemented in murasaki::Adau1361.

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

send one command to CODEC

Parameters

command	command data array.
size	command length by [byte].

Implemented in murasaki::Adau1361.

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

Set channel gain.

Parameters

channel	
left_gain	
right_gain	

Implemented in murasaki::Adau1361.

12.2.3.4 virtual void murasaki::AudioCodecStrategy::Start (void) [pure virtual]

Actual initializer.

Initialize the codec itself and start the conversion process. and configure for given parameter.

Finally, set the input gain to 0dB.

Implemented in murasaki::Adau1361.

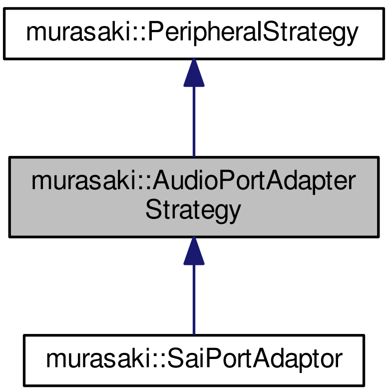
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/audiocodecstrategy.hpp

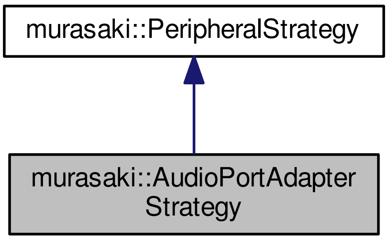
12.3 murasaki::AudioPortAdapterStrategy Class Reference

`#include <audioportadapterstrategy.hpp>`

Inheritance diagram for murasaki::AudioPortAdapterStrategy:



Collaboration diagram for murasaki::AudioPortAdapterStrategy:



Public Member Functions

- virtual void StartTransferTx (uint8_t ∗tx_buffer, unsigned int channel_len)=0
- virtual void StartTransferRx (uint8_t ∗rx_buffer, unsigned int channel_len)=0
- virtual unsigned int GetNumberOfDMAPhase ()=0
- virtual unsigned int GetNumberOfChannelsTx ()=0
- virtual unsigned int GetSampleWordSizeTx ()=0
- virtual unsigned int GetNumberOfChannelsRx ()=0
- virtual unsigned int GetSampleWordSizeRx ()=0
- virtual unsigned int DetectPhase (unsigned int phase)
- virtual bool HandleError (void ∗ptr)=0
- virtual bool Match (void ∗peripheral_handle)=0
- virtual void ∗ GetPeripheralHandle ()=0

Additional Inherited Members

12.3.1 Detailed Description

Template class of the audio device adaptor.

12.3.2 Member Function Documentation

12.3.2.1 virtual unsigned int murasaki::AudioPortAdapterStrategy::DetectPhase (unsigned int *phase*) [inline], [virtual]

DMA phase detector.

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.

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

Return how many channels are in the transfer.

Returns

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

Implemented in murasaki::SaiPortAdaptor.

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

Return how many channels are in the transfer.

Returns

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

Implemented in murasaki::SaiPortAdaptor.

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

Return how many DMA phase is implemented.

Returns

2 for Double buffer, 3 for Tripple buffer.

Implemented in murasaki::SaiPortAdaptor.

12.3.2.5 virtual void∗ murasaki::AudioPortAdapterStrategy::GetPeripheralHandle ( ) [pure virtual]

pass the raw peripheral handler

Returns

pointer to the raw peripheral handler hidden in a class.

Implements murasaki::PeripheralStrategy.

Implemented in murasaki::SaiPortAdaptor.

12.3.2.6 virtual unsigned int murasaki::AudioPortAdapterStrategy::GetSampleWordSizeRx ( ) [pure virtual]

Return the size of the one sample.

Returns

2 or 4. The unit is [Byte]

Implemented in murasaki::SaiPortAdaptor.

12.3.2.7 virtual unsigned int murasaki::AudioPortAdapterStrategy::GetSampleWordSizeTx ( ) [pure virtual]

Return the size of the one sample.

Returns

2 or 4. The unit is [Byte]

Implemented in murasaki::SaiPortAdaptor.

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

Handling error report of device.

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::SaiPortAdaptor.

12.3.2.9 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::SaiPortAdaptor.

12.3.2.10 virtual void murasaki::AudioPortAdapterStrategy::StartTransferRx (uint8_t * *rx_buffer,* unsigned int *channel_len*) [pure virtual]

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 transfer on the circular buffer once after it starts.

Implemented in murasaki::SaiPortAdaptor.

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

Kick start routine to start the TX 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 transfer on the circular buffer once after it starts.

Implemented in murasaki::SaiPortAdaptor.

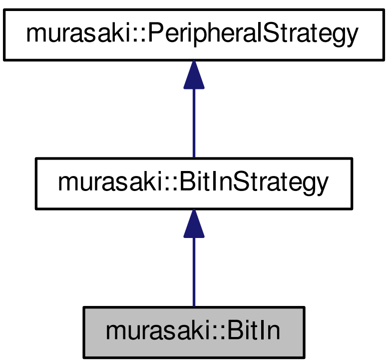
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/audioportadapterstrategy.hpp

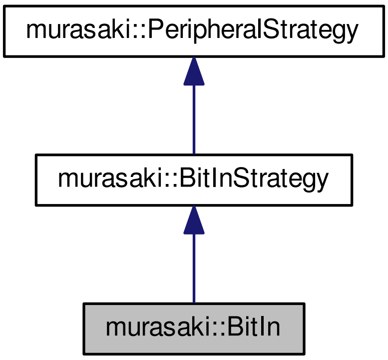
12.4 murasaki::BitIn Class Reference

`#include <bitin.hpp>`

Inheritance diagram for murasaki::BitIn:



Collaboration diagram for murasaki::BitIn:



Public Member Functions

- BitIn (GPIO_TypeDef ∗port, uint16_t pin)
- virtual unsigned int Get (void)
- virtual void ∗ GetPeripheralHandle ()

Additional Inherited Members

12.4.1 Detailed Description

The BitIn class is the wrapper of the GPIO controller. To use the BitIn class, make an instance with GPIO_TypeDef ∗ type pointer. For example, to create an instance for a switch peripheral:

```
my_swithc = new murasaki::BitIn(sw_port, sw_pin);
```

Where sw_port and sw_pin are the macro generated by CubeIDE for GPIO pin. the GPIO peripheral have to be configured to be right direction.

12.4.2 Constructor & Destructor Documentation

12.4.2.1 murasaki::BitIn::BitIn (GPIO_TypeDef ∗ *port*, uint16_t *pin*)

Constructor.

Parameters

port	Pinter to the port strict.
pin	Number of the pin to input.

12.4.3 Member Function Documentation

12.4.3.1 unsigned int murasaki::BitIn::Get (void) [virtual]

Get a status of the output pin.

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::BitInStrategy.

12.4.3.2 void ∗ murasaki::BitIn::GetPeripheralHandle ( ) [virtual]

pass the raw peripheral handler

Returns

pointer to the GPIO_type variable hidden in a class.

Implements murasaki::PeripheralStrategy.

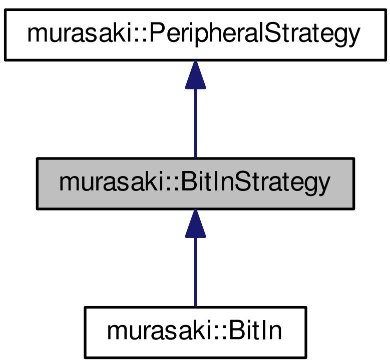
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitin.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/bitin.cpp

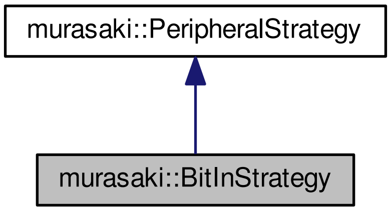
12.5 murasaki::BitInStrategy Class Reference

`#include <bitinstrategy.hpp>`

Inheritance diagram for murasaki::BitInStrategy:



Collaboration diagram for murasaki::BitInStrategy:



Public Member Functions

- virtual unsigned int Get (void)=0

Additional Inherited Members

12.5.1 Detailed Description

A prototype of the general purpose bit input class

12.5.2 Member Function Documentation

12.5.2.1 virtual unsigned int murasaki::BitInStrategy::Get (void) [pure virtual]

Get a status of the input pin.

Returns

1 or 0 as output state.

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

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

Implemented in murasaki::BitIn.

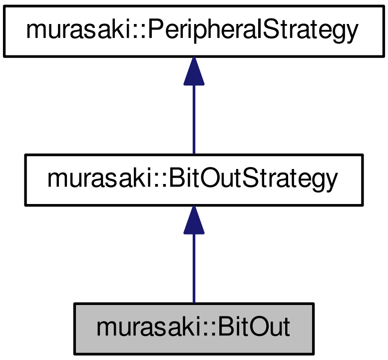
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitinstrategy.hpp

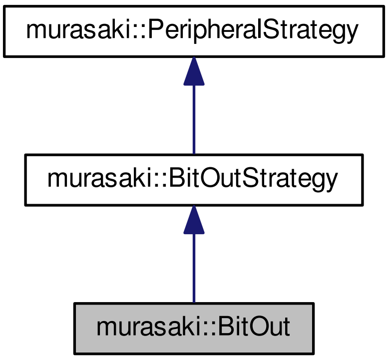
12.6 murasaki::BitOut Class Reference

`#include <bitout.hpp>`

Inheritance diagram for murasaki::BitOut:



Collaboration diagram for murasaki::BitOut:



Public Member Functions

- BitOut (GPIO_TypeDef ∗port, uint16_t pin)
- virtual void Set (unsigned int state=1)
- virtual unsigned int Get (void)
- virtual void ∗ GetPeripheralHandle ()

Additional Inherited Members

12.6.1 Detailed Description

The BitOut class is the wrapper of the GPIO controller. To use the BitOut class, make an instance with GPIO_↩ TypeDef ∗ type pointer. For example, to create an instance for the a peripheral:

```
my_LED = new murasaki::BitOut(LED_port, LED_pin);
```

Where LED_port and LED_pin are the macro generated by CubeIDE for GPIO pin. the GPIO peripheral have to be configured to be right direction.

12.6.2 Constructor & Destructor Documentation

12.6.2.1 murasaki::BitOut::BitOut (GPIO_TypeDef ∗ *port*, uint16_t *pin*)

Constructor.

Parameters

port	Pinter to the port strict.
pin	Number of the pin to output.

12.6.3 Member Function Documentation

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

Get a status of the output pin.

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.

12.6.3.2 void ∗ murasaki::BitOut::GetPeripheralHandle ( ) [virtual]

pass the raw peripheral handler

Returns

pointer to the GPIO_type variable hidden in a class.

Implements murasaki::PeripheralStrategy.

12.6.3.3 void murasaki::BitOut::Set (unsigned int *state* = 1) [virtual]

Set a status of the output pin.

Parameters

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

Implements murasaki::BitOutStrategy.

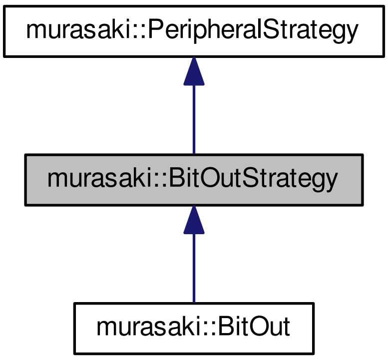
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitout.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/bitout.cpp

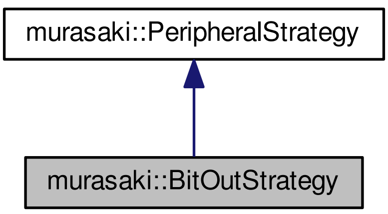
12.7 murasaki::BitOutStrategy Class Reference

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

Inheritance diagram for murasaki::BitOutStrategy:



Collaboration diagram for murasaki::BitOutStrategy:



Public Member Functions

- virtual void Set (unsigned int state=1)=0
- virtual unsigned int Get (void)=0

Additional Inherited Members

12.7.1 Detailed Description

A prototype of the general purpose bit out class

12.7.2 Member Function Documentation

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

Get a status of the output pin.

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.

12.7.2.2 virtual void murasaki::BitOutStrategy::Set (unsigned int *state* = 1) [pure virtual]

Set a status of the output pin.

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:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitoutstrategy.hpp

12.8 murasaki::CriticalSection Class Reference

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

Public Member Functions

- void Enter ()
- void Leave ()

12.8.1 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().

12.8.2 Member Function Documentation

12.8.2.1 void murasaki::CriticalSection::Enter ( )

Entering critical section.

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

12.8.2.2 void murasaki::CriticalSection::Leave ( )

Leaving crititical section.

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

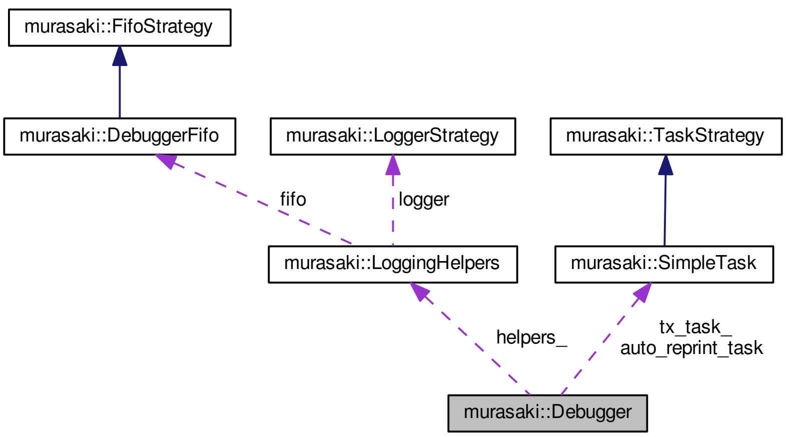
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/criticalsection.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/criticalsection.cpp

12.9 murasaki::Debugger Class Reference

```
#include <debugger.hpp>
```

Collaboration diagram for murasaki::Debugger:



Public Member Functions

- Debugger (LoggerStrategy ∗logger)
- void Printf (const char ∗fmt,...)
- char GetchFromTask ()
- void RePrint ()
- void AutoRePrint ()

Protected Attributes

- char line_ [PLATFORM_CONFIG_DEBUG_LINE_SIZE]
- murasaki::SyslogSeverity severity_
- uint32_t facility_mask_

12.9.1 Detailed Description

Wrapper class to help the printf debug. The printf() method 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.

12.9.2 Constructor & Destructor Documentation

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

Constructor. Create internal variable.

Parameters

logger	The pointer to the LoggerStrategy wrapper class variable.

12.9.3 Member Function Documentation

12.9.3.1 void murasaki::Debugger::AutoRePrint ( )

Print history automatically.

Once this member function is called, internally new task is created. This new task watches input by GetchFromTask() and for each input char is recevied, trigger the RePrint().

This auto reprint funciton is exclusive and 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.

12.9.3.2 char murasaki::Debugger::GetchFromTask ( )

Receive one character from serial port.

Returns

Received character.

A blcoking 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.

Becareful, this is synchronous and blocking while the Debug::Printf() is asynchronous and non-blocking.

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

Debug output function.

Parameters

fmt	Format string
...	optional parameters

The printf() compatible method. This method can be called from both task context and ISR context. This method 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 method streos as possible, and discard the rest of string. That mean, this method is neither synchronous nor blocking.

This member function is non-blocking, non-asynchronous, thread safe and re-entrant.

At 2018/Jan/14 measurement, task stack was consumed 49bytes.

12.9.3.4 void murasaki::Debugger::RePrint ( )

Print the old data again.

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-setn is the one in the data in side circular buffer. Then, the resent size is same as PLATFORM_CONFIG_DEBUG_BUFFER_SIZE .

12.9.4 Member Data Documentation

12.9.4.1 uint32_t murasaki::Debugger::facility_mask_ [protected]

Syslog facility filter mask.

If certain bit is "1", the corresponding Syslog facility is allowed to output. By default the value is 0xFFFF (equivalent to SyslogAllowAllFacilities(0xFFFFFFFF))

12.9.4.2 char murasaki::Debugger::line_[PLATFORM_CONFIG_DEBUG_LINE_SIZE] [protected]

as receiver for the snprintf()

This variable can be local variable of the printf() member function. In thiss 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.

12.9.4.3 murasaki::SyslogSeverity murasaki::Debugger::severity_ [protected]

Syslog severity threshold.

All seveirity level lower than this value will be ignored by Syslog() function. Note that murasaki::kseEmergency is the highest and murasaki::kseDebug is the lowerest seveirty.

By default, the severity level threshold is murasaki::kseError. That mean, the weaker severity than kseError is ignored.

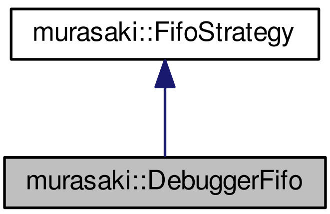
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/debugger.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/debugger.cpp

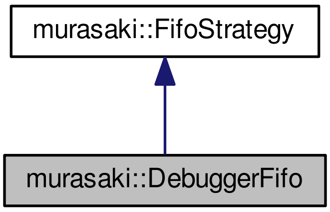
12.10 murasaki::DebuggerFifo Class Reference

```
#include <debuggerfifo.hpp>
```

Inheritance diagram for murasaki::DebuggerFifo:



Collaboration diagram for murasaki::DebuggerFifo:



Public Member Functions

- DebuggerFifo (unsigned int buffer_size)
- virtual unsigned int Get (uint8_t data[ ], unsigned int size)
- virtual void SetPostMortem ()

12.10.1 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 funciton returns with "copied" data count and data. If the internal buffer is empty, it returns without copy data.

12.10.2 Constructor & Destructor Documentation

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

Create an internal buffer.

Parameters

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

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

12.10.3 Member Function Documentation

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

Get the data from the internal buffer. This is thread safe function. Do not call from ISR.

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.

12.10.3.2 void murasaki::DebuggerFifo::SetPostMortem ( ) [virtual]

Transit to the post mortem mode.

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

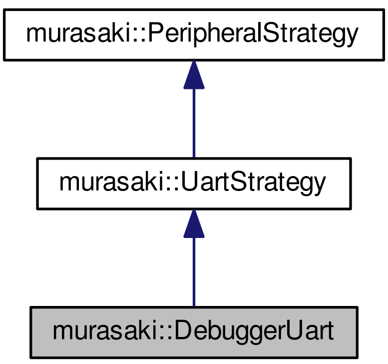
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/debuggerfifo.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/debuggerfifo.cpp

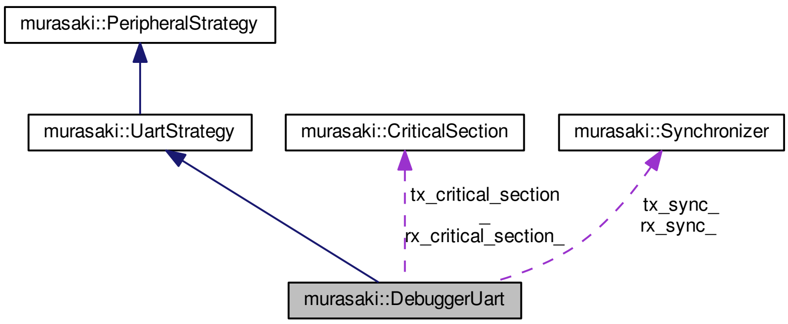
12.11 murasaki::DebuggerUart Class Reference

```
#include <debuggeruart.hpp>
```

Inheritance diagram for murasaki::DebuggerUart:



Collaboration diagram for murasaki::DebuggerUart:



Public Member Functions

- DebuggerUart (UART_HandleTypeDef ∗uart)
- virtual void SetHardwareFlowControl (UartHardwareFlowControl control)
- virtual void SetSpeed (unsigned int baud_rate)
- virtual murasaki::UartStatus Transmit (const uint8_t ∗data, unsigned int size, unsigned int timeout_ms)
- virtual murasaki::UartStatus Receive (uint8_t ∗data, unsigned int count, unsigned int ∗transfered_count, UartTimeout uart_timeout, unsigned int timeout_ms)
- virtual bool TransmitCompleteCallback (void ∗const ptr)
- virtual bool ReceiveCompleteCallback (void ∗const ptr)
- virtual bool HandleError (void ∗const ptr)

Additional Inherited Members

12.11.1 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::TransmitCompleteCallback() 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.

12.11.2 Constructor & Destructor Documentation

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

Constructor.

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.

12.11.3 Member Function Documentation

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

Error handling.

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.

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

Receive raw data through an UART by synchronous mode.

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.

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

Call back for entire block transfer completion.

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.

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

Set the behavior of the hardware flow control.

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.

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

Set the BAUD rate.

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.

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

Transmit raw data through an UART by synchronous mode.

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.

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

Call back for entire block transfer completion.

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:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/debuggeruart.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/debuggeruart.cpp

12.12 murasaki::DuplexAudio Class Reference

```
#include <duplexaudio.hpp>
```

Public Member Functions

- DuplexAudio (murasaki::AudioPortAdapterStrategy ∗peripheral_adapter, unsigned int channel_length)
- void TransmitAndReceive (float ∗tx_left, float ∗tx_right, float ∗rx_left, float ∗rx_right)
- void TransmitAndReceive (float ∗∗tx_channels, float ∗∗rx_channels, unsigned int tx_num_of_channels, unsigned int rx_num_of_channels)
- bool DmaCallback (void ∗peripheral, unsigned int phase)
- virtual bool HandleError (void ∗peripheral)

12.12.1 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 interface with application.
- data range is [-1.0, 1.0) as interface with application.
- blocking and synchronous API
- Internal DMA operation.

Note : this class assumes the Fs of the TX and RX are same and both Tx and RX are fully synchronized.

Internally, this class provides a multi-buffers DMA operation between the audio peripheral and caller algorithm. The key API is the TransmitAndReceive() member function. This function provides the 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, user doesn't need to care about above things.

Because of the complicated audio data structure, there are several terminologies which 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 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 buffere is called as phase. For example, the double buffere DMA can be phase 0 or 1 and incremented as modulo 2.

The number of phase is specified to the constructor, by programmer. This phase have to be aligned with hardware.

12.12.2 Constructor & Destructor Documentation

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

Constructor.

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 channel named left and right.

12.12.3 Member Function Documentation

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

Callback function on the RX DMA interrupt.

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.

This function returns if peripheral parameter is match with the one passed by the constructor.

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

Call this function from the interrupt handler.

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.

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

Stereo audio transmission/receiving.

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

Blocking and 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.

Following is the typoical 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
    ...
}
```

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

Multi channel audio transmission/receiving.

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.

Infrastructure function for the public functions.

Blocking and 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 the common base for the other 2 public TransmitAndRecieve(). To serve both of them, this function receives the number of channels explictly.

```
#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
    ...
}
```

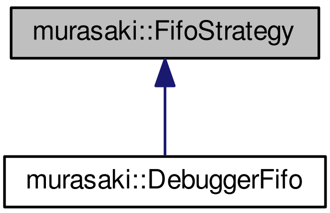
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/duplexaudio.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/duplexaudio.cpp

12.13 murasaki::FifoStrategy Class Reference

```
#include <fifostrategy.hpp>
```

Inheritance diagram for murasaki::FifoStrategy:



Public Member Functions

- FifoStrategy (unsigned int buffer_size)
- virtual unsigned int Put (uint8_t const data[ ], unsigned int size)
- virtual unsigned int Get (uint8_t data[ ], unsigned int size)

12.13.1 Detailed Description

Foundemental 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 funciton returns with "copied" data count and data. If the internal buffer is empty, it returns without copy data.

12.13.2 Constructor & Destructor Documentation

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

Create an internal buffer.

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.

12.13.3 Member Function Documentation

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

Get the data from the internal buffer.

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.

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

Put the data into the internal buffer.

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:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/fifostrategy.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/fifostrategy.cpp

12.14 murasaki::GPIO_type Struct Reference

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

12.14.1 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 funciton.

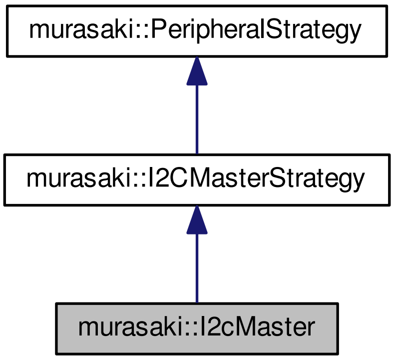
The documentation for this struct was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitout.hpp

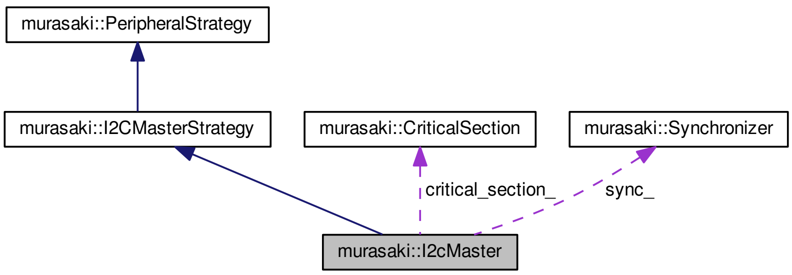
12.15 murasaki::I2cMaster Class Reference

`#include <i2cmaster.hpp>`

Inheritance diagram for murasaki::I2cMaster:



Collaboration diagram for murasaki::I2cMaster:



Public Member Functions

- I2cMaster (I2C_HandleTypeDef ∗i2c_handle)
- virtual murasaki::I2cStatus Transmit (unsigned int addrs, const uint8_t ∗tx_data, unsigned int tx_size, unsigned int ∗transfered_count, unsigned int timeout_ms)
- virtual murasaki::I2cStatus Receive (unsigned int addrs, uint8_t ∗rx_data, unsigned int rx_size, unsigned int ∗transfered_count, unsigned int timeout_ms)
- 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)
- virtual bool TransmitCompleteCallback (void ∗ptr)
- virtual bool ReceiveCompleteCallback (void ∗ptr)
- virtual bool HandleError (void ∗ptr)

Additional Inherited Members

12.15.1 Detailed Description

The I2cMaster class is the wrapper of the I2C controller. To use the I2cMaster class, make 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. To use this class, the I2C peripheral have to be configured to use the interrupt functionality without DMA. The bitrate should be configured by the CubeIDE.

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

```
void HAL_I2C_TxCpltCallback(I2C_HandleTypeDef * hi2c)
{
    my_i2c3->TransmitCompleteCallback(hi2c);
}
```

Where HAL_I2C_TxCpltCallback is a predefined name of the I2C interrupt handler. This is invoked by system whenever a interrupt baed I2C 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 I2Cn where n is 1, 2, 3... To avoid the confusion, I2cMaster::Transmit←CompleteCallback() method chckes whether given parameter matches with its I2C_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_I2C_TxCpltCallback().

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 timeout_ms parameter. By default, this parameter is set by kwmsIndefinitely which specifes never time out.

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 kwmsIndefinitely which species never time out.

The I2cMaster::TransmitThenReceive() member function is synchronous function. A programmer can specify the timeout by timeout_ms parameter. By default, this parameter is set by kwmsIndefinitely which species 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.

Note : In case an time out occurs during transmit / receive, this implementation calls HAL_I2C_MASTER_ABOR←T_IT(). But it is unknown whether this is 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. And also, it doesn't tell what's happen if the HAL_I2C_MASTER_ABORT_IT() is called.

According to the source code of the HAL_I2C_MASTER_ABORT_IT(), no interrupt will be raised by this API call.

12.15.2 Constructor & Destructor Documentation

12.15.2.1 murasaki::I2cMaster::I2cMaster (I2C_HandleTypeDef * *i2c_handle*)

Constructor.

Parameters

i2c_handle	Peripheral handle created by CubeMx

12.15.3 Member Function Documentation

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

Error handling.

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.

Implements murasaki::I2CMasterStrategy.

12.15.3.2 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]

Thread safe, synchronous receiving over I2C.

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. 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::I2CMasterStrategy.

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

Call back to be called for entire block transfer is complete.

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.

Implements murasaki::I2CMasterStrategy.

12.15.3.4 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]

Thread safe, synchronous transmission over I2C.

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. 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::I2CMasterStrategy.

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

Call back to be called notify the transfer is complete.

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.

Implements murasaki::I2CMasterStrategy.

12.15.3.6 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]

Thread safe, synchronous transmission and then receiving over I2C.

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 transmit the data, and the, by repeated start function, it receives data. The transmission device address and receiving device address is same.

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 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.

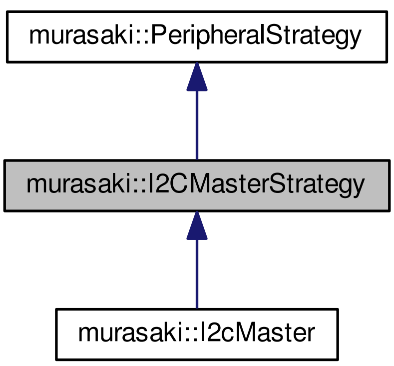
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/i2cmaster.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/i2cmaster.cpp

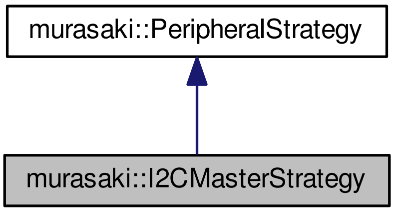
12.16 murasaki::I2CMasterStrategy Class Reference

```
#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
- 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
- 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
- virtual bool TransmitCompleteCallback (void ∗ptr)=0
- virtual bool ReceiveCompleteCallback (void ∗ptr)=0
- virtual bool HandleError (void ∗ptr)=0

Additional Inherited Members

12.16.1 Detailed Description

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 mean, until the transmit / receive terminates, both method doesn't return.

Two call back member functions are prepared to sync with the interrupt which tells the end of Transmit/Receive.

12.16.2 Member Function Documentation

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

Handling error report of device.

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::I2cMaster.

12.16.2.2 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]`

Thread safe, synchronous receiving over I2C.

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 should be internally exclusive between multiple task access. In other word, it should be thread save.

Implemented in murasaki::I2cMaster.

12.16.2.3 virtual bool murasaki::I2CMasterStrategy::ReceiveCompleteCallback (void * *ptr*) `[pure virtual]`

Call back to be called for entire block transfer is complete.

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.

12.16.2.4 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]

Thread safe, synchronous transmission over I2C.

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 should be internally exclusive between multiple task access. In other word, it should be thread save.

Implemented in murasaki::I2cMaster.

12.16.2.5 virtual bool murasaki::I2CMasterStrategy::TransmitCompleteCallback (void * *ptr*) [pure virtual]

Call back to be called notify the transfer is complete.

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.

12.16.2.6 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]

Thread safe, synchronous transmission and then receiving over I2C.

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 transmit the data, and the, by repeated start function, it receives data. The transmission device address and receiving device address is same.

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.

Implemented in murasaki::I2cMaster.

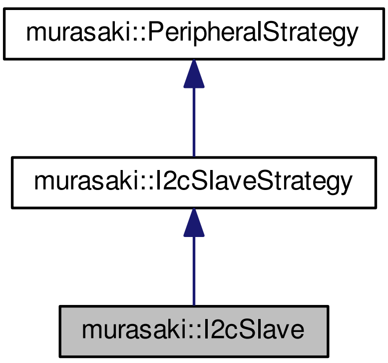
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/i2cmasterstrategy.hpp

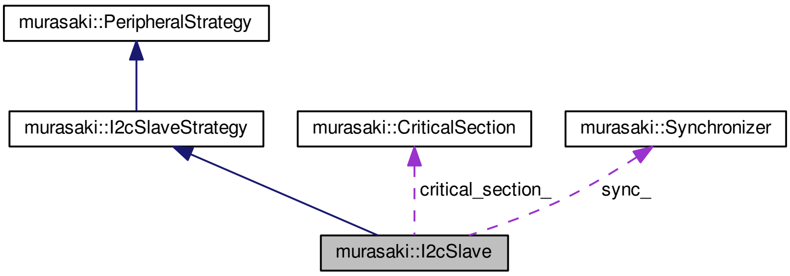
12.17 murasaki::I2cSlave Class Reference

```
#include <i2cslave.hpp>
```

Inheritance diagram for murasaki::I2cSlave:



Collaboration diagram for murasaki::I2cSlave:



Public Member Functions

- virtual murasaki::I2cStatus Transmit (const uint8_t ∗tx_data, unsigned int tx_size, unsigned int ∗transfered_count, unsigned int timeout_ms)
- virtual murasaki::I2cStatus Receive (uint8_t ∗rx_data, unsigned int rx_size, unsigned int ∗transfered_count, unsigned int timeout_ms)
- virtual bool TransmitCompleteCallback (void ∗ptr)
- virtual bool ReceiveCompleteCallback (void ∗ptr)
- virtual bool HandleError (void ∗ptr)

Additional Inherited Members

12.17.1 Detailed Description

The I2cSlave class is the wrapper of the I2C controller. To use the I2cSlave class, make 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. To use this class, the I2C peripheral have to be configured to use the interrupt functionality without DMA. The bit rate and the peripheral address should be configured by the CubeIDE.

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

```
void HAL_I2C_TxCpltCallback(I2C_HandleTypeDef * hi2c)
{
   if ( my_i2c3->TransmitCompleteCallback(hi2c))
     return;
}

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

}
```

Where HAL_I2C_TxCpltCallback is a predefined name of the I2C interrupt handler. This is invoked by system whenever a interrupt baed I2C transmission is complete. Because the default function is weakly bound, above definition will override the default one.

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

As same as Tx, RX needs HAL_I2C_TxCpltCallback().

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 specifes never time out.

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 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.

- Note : In case an time out occurs during transmit / receive, this implementation calls HAL_I2C_DeInit()/H←AL_I2C_Init(). But it is unknown whether this is 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.

12.17.2 Member Function Documentation

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

Error handling.

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.

Implements murasaki::I2cSlaveStrategy.

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

Thread safe, synchronous receiving over I2C.

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 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 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.

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

Call back to be called for entire block transfer is complete.

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.

Implements murasaki::I2cSlaveStrategy.

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

Thread safe, synchronous transmission over I2C.

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 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 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.

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

Call back to be called notify the transfer is complete.

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.

Implements murasaki::I2cSlaveStrategy.

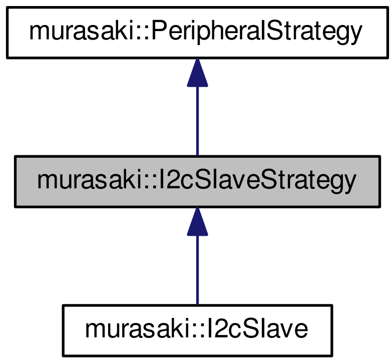
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/i2cslave.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/i2cslave.cpp

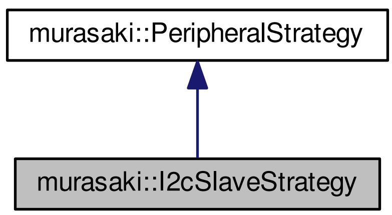
12.18 murasaki::I2cSlaveStrategy Class Reference

`#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
- virtual murasaki::I2cStatus Receive (uint8_t ∗rx_data, unsigned int rx_size, unsigned int ∗transfered_count=nullptr, unsigned int timeout_ms=murasaki::kwmsIndefinitely)=0
- virtual bool TransmitCompleteCallback (void ∗ptr)=0
- virtual bool ReceiveCompleteCallback (void ∗ptr)=0
- virtual bool HandleError (void ∗ptr)=0

Additional Inherited Members

12.18.1 Detailed Description

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.

Two call back member functions are prepared to sync with the interrupt which tells the end of Transmit/Receive.

12.18.2 Member Function Documentation

12.18.2.1 virtual bool murasaki::I2cSlaveStrategy::HandleError (void * *ptr*) [pure virtual]

Handling error report of device.

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::I2cSlave.

12.18.2.2 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]

Thread safe, synchronous receiving over I2C.

Parameters

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 should be internally exclusive between multiple task access. In other word, it should be thread save.

Implemented in murasaki::I2cSlave.

12.18.2.3 virtual bool murasaki::I2cSlaveStrategy::ReceiveCompleteCallback (void * *ptr*) [pure virtual]

Call back to be called for entire block transfer is complete.

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::I2cSlave.

12.18.2.4 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]

Thread safe, synchronous transmission over I2C.

Parameters

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 should be internally exclusive between multiple task access. In other word, it should be thread save.

Implemented in murasaki::I2cSlave.

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

Call back to be called notify the transfer is complete.

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::I2cSlave.

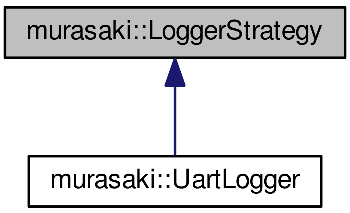
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/i2cslavestrategy.hpp

12.19 murasaki::LoggerStrategy Class Reference

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

Inheritance diagram for murasaki::LoggerStrategy:



Public Member Functions

- virtual ∼LoggerStrategy ()
- virtual void putMessage (char message[ ], unsigned int size)=0
- virtual char getCharacter ()=0
- virtual void DoPostMortem (void ∗debugger_fifo)

12.19.1 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.

12.19.2 Constructor & Destructor Documentation

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

Detructor.

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

12.19.3 Member Function Documentation

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

Start post mortem process.

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 implments a method, it should call the Debugger::SetPost←Mortem() internaly.

Reimplemented in murasaki::UartLogger.

12.19.3.2 virtual char murasaki::LoggerStrategy::getCharacter ( ) [pure virtual]

Character input member function.

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.

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

Message output member function.

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 blcoking. That mean, it will not wayt until data is stored to the storage or output.

Implemented in murasaki::UartLogger.

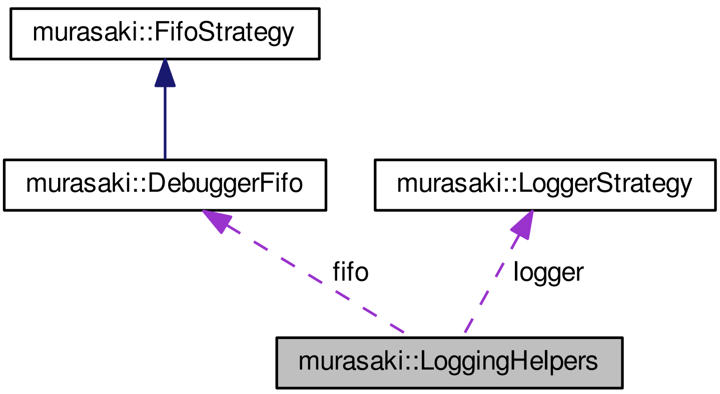
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/loggerstrategy.hpp

12.20 murasaki::LoggingHelpers Struct Reference

`#include <debuggerfifo.hpp>`

Collaboration diagram for murasaki::LoggingHelpers:



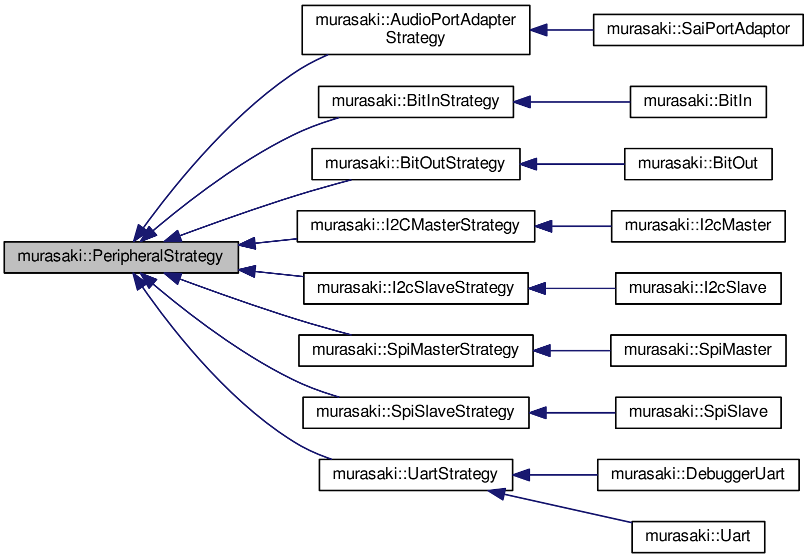
The documentation for this struct was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/debuggerfifo.hpp

12.21 murasaki::PeripheralStrategy Class Reference

`#include <peripheralstrategy.hpp>`

Inheritance diagram for murasaki::PeripheralStrategy:



Public Member Functions

- virtual bool Match (void ∗peripheral_handle)

Protected Member Functions

- virtual void ∗ GetPeripheralHandle ()=0

12.21.1 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.

12.21.2 Member Function Documentation

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

pass the raw peripheral handler

Returns

pointer to the raw peripheral handler hidden in a class.

Implemented in murasaki::SaiPortAdaptor, murasaki::AudioPortAdapterStrategy, murasaki::BitOut, and murasaki::BitIn.

12.21.2.2 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::SaiPortAdaptor, and murasaki::AudioPortAdapterStrategy.

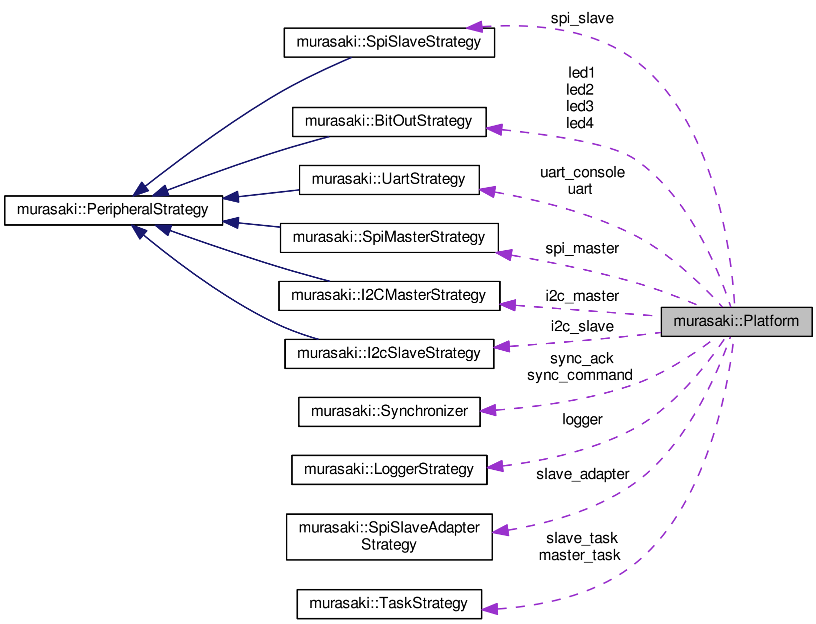
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/peripheralstrategy.hpp

12.22 murasaki::Platform Struct Reference

`#include <platform_defs.hpp>`

Collaboration diagram for murasaki::Platform:



12.22.1 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

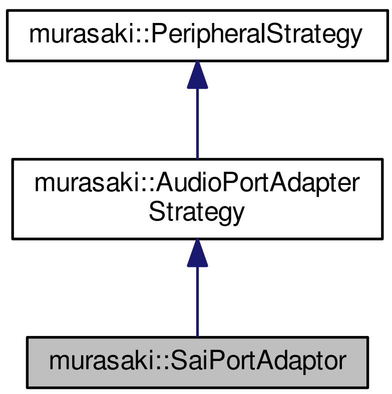
The documentation for this struct was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Inc/platform_defs.hpp

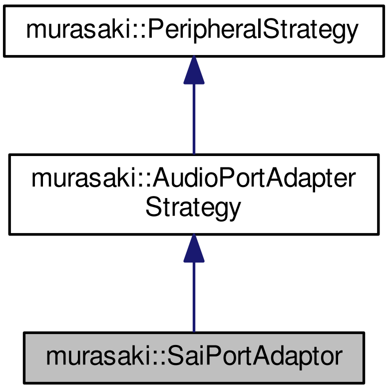
12.23 murasaki::SaiPortAdaptor Class Reference

```
#include <saiportadaptor.hpp>
```

Inheritance diagram for murasaki::SaiPortAdaptor:



Collaboration diagram for murasaki::SaiPortAdaptor:



Public Member Functions

- SaiPortAdaptor (SAI_HandleTypeDef ∗tx_peripheral, SAI_HandleTypeDef ∗rx_peripheral)
- virtual void StartTransferTx (uint8_t ∗tx_buffer, unsigned int channel_len)
- virtual void StartTransferRx (uint8_t ∗rx_buffer, unsigned int channel_len)
- virtual unsigned int GetNumberOfDMAPhase ()
- virtual unsigned int GetNumberOfChannelsTx ()
- virtual unsigned int GetSampleWordSizeTx ()
- virtual unsigned int GetNumberOfChannelsRx ()
- virtual unsigned int GetSampleWordSizeRx ()
- virtual bool HandleError (void ∗ptr)
- virtual bool Match (void ∗peripheral_handle)
- virtual void ∗ GetPeripheralHandle ()

Additional Inherited Members

12.23.1 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 misaligned configuration gives broken audio.

12.23.2 Constructor & Destructor Documentation

12.23.2.1 murasaki::SaiPortAdaptor::SaiPortAdaptor (SAI_HandleTypeDef ∗ *tx_peripheral,* SAI_HandleTypeDef ∗ *rx_peripheral*)

Constructor.

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.

12.23.3 Member Function Documentation

12.23.3.1 unsigned int murasaki::SaiPortAdaptor::GetNumberOfChannelsRx ( ) [virtual]

Return how many channels are in the transfer.

Returns

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

Implements murasaki::AudioPortAdapterStrategy.

12.23.3.2 unsigned int murasaki::SaiPortAdaptor::GetNumberOfChannelsTx ( ) [virtual]

Return how many channels are in the transfer.

Returns

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

Implements murasaki::AudioPortAdapterStrategy.

12.23.3.3 virtual unsigned int murasaki::SaiPortAdaptor::GetNumberOfDMAPhase ( ) [inline], [virtual]

Return how many DMA phase is implemented.

Returns

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

Implements murasaki::AudioPortAdapterStrategy.

12.23.3.4 void * murasaki::SaiPortAdaptor::GetPeripheralHandle ( ) [virtual]

pass the raw peripheral handler

Returns

pointer to the raw peripheral handler hidden in a class.

Implements murasaki::AudioPortAdapterStrategy.

12.23.3.5 unsigned int murasaki::SaiPortAdaptor::GetSampleWordSizeRx ( ) [virtual]

Return the size of the one sample.

Returns

2 or 4. The unit is [Byte]

Implements murasaki::AudioPortAdapterStrategy.

12.23.3.6 unsigned int murasaki::SaiPortAdaptor::GetSampleWordSizeTx ( ) [virtual]

Return the size of the one sample.

Returns

2 or 4. The unit is [Byte]

Implements murasaki::AudioPortAdapterStrategy.

12.23.3.7 bool murasaki::SaiPortAdaptor::HandleError (void * *ptr*) [virtual]

Handling error report of device.

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.

12.23.3.8 bool murasaki::SaiPortAdaptor::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.

12.23.3.9 void murasaki::SaiPortAdaptor::StartTransferRx (uint8_t * *rx_buffer*, unsigned int *channel_len*) [virtual]

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 transfer on the circular buffer once after it starts.

Implements murasaki::AudioPortAdapterStrategy.

12.23.3.10 void murasaki::SaiPortAdaptor::StartTransferTx (uint8_t * *tx_buffer*, unsigned int *channel_len*) [virtual]

Kick start routine to start the TX 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 transfer on the circular buffer once after it starts.

Implements murasaki::AudioPortAdapterStrategy.

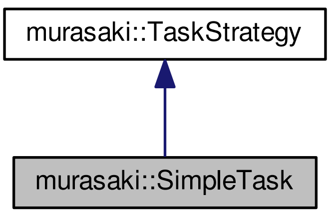
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/saiportadaptor.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/saiportadaptor.cpp

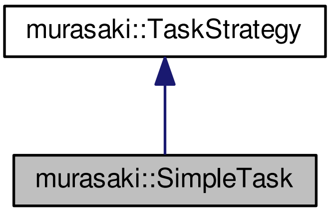
12.24 murasaki::SimpleTask Class Reference

```
#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 ∗))

Protected Member Functions

- virtual void TaskBody (const void ∗ptr)

Additional Inherited Members

12.24.1 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();
```

12.24.2 Constructor & Destructor Documentation

12.24.2.1 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*)

Ease to use task class.

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.

12.24.3 Member Function Documentation

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

Task member function.

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.

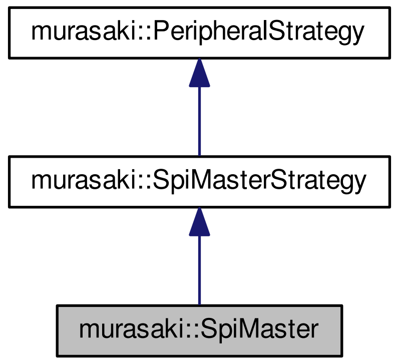
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/simpletask.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/simpletask.cpp

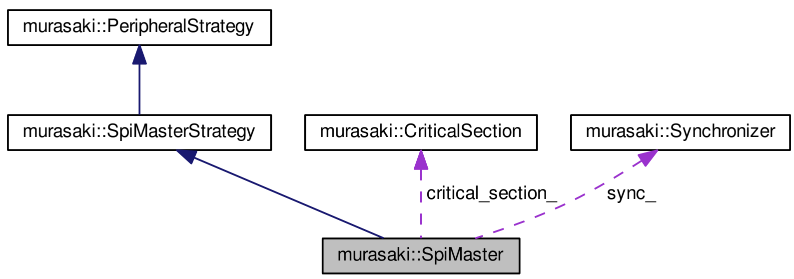
12.25 murasaki::SpiMaster Class Reference

`#include <spimaster.hpp>`

Inheritance diagram for murasaki::SpiMaster:



Collaboration diagram for murasaki::SpiMaster:



Public Member Functions

- SpiMaster (SPI_HandleTypeDef ∗spi_handle)
- 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)
- virtual bool TransmitAndReceiveCompleteCallback (void ∗ptr)
- virtual bool HandleError (void ∗ptr)

Additional Inherited Members

12.25.1 Detailed Description

The SpiMaster class is the wrapper of the SPI controller. To use the SpiMaster class, make 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. To use this class, the SPI peripheral have to be configured to use the interrupt and DMA. The bitrate 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. 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.

Once the instance and callbacks 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 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.

Note : The behavior of when the timeout happen is not tested. Actually, it should not happen because DMA is taken in SPI transmission. Murasaki stpos 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.

12.25.2 Constructor & Destructor Documentation

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

Constractor.

Parameters

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

12.25.3 Member Function Documentation

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

Error handling.

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.

Implements murasaki::SpiMasterStrategy.

12.25.3.2 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]

Data transfer to/from SPI slave.

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.

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

Callback to notify the end of transfer.

Parameters

ptr	Pointer to the control object.

Returns

true if no error.

Implements murasaki::SpiMasterStrategy.

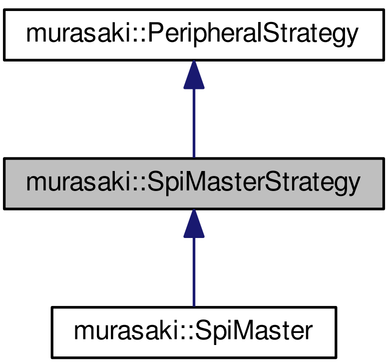
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spimaster.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/spimaster.cpp

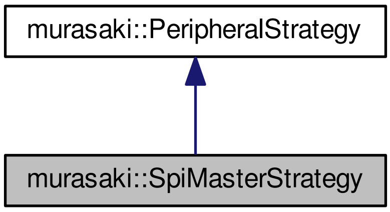
12.26 murasaki::SpiMasterStrategy Class Reference

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

Additional Inherited Members

12.26.1 Detailed Description

This class provides a thread safe, synchronous SPI transfer.

12.26.2 Member Function Documentation

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

Handling error report of device.

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.

12.26.2.2 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]

Thread safe, synchronous SPI transfer.

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.

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

Callback to notifiy the end of transfer.

Parameters

ptr	Pointer to the control object.

Returns

true if no error.

Implemented in murasaki::SpiMaster.

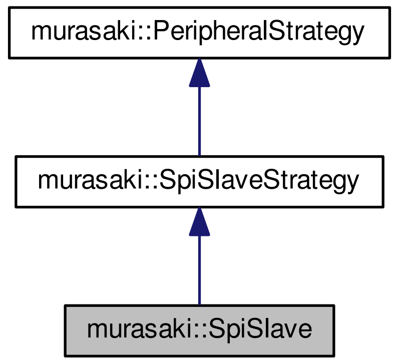
The documentation for this class was generated from the following file:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spimasterstrategy.hpp

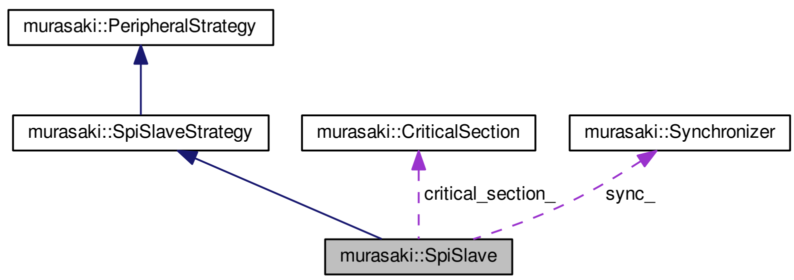
12.27 murasaki::SpiSlave Class Reference

```
#include <spislave.hpp>
```

Inheritance diagram for murasaki::SpiSlave:



Collaboration diagram for murasaki::SpiSlave:



Public Member Functions

- SpiSlave (SPI_HandleTypeDef ∗spi_handle)
- 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)
- virtual bool TransmitAndReceiveCompleteCallback (void ∗ptr)
- virtual bool HandleError (void ∗ptr)

Additional Inherited Members

12.27.1 Detailed Description

The SpiSlave class is the wrapper of the SPI controller. 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. 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.

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 methods 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.

12.27.2 Constructor & Destructor Documentation

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

Constractor.

Parameters

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

12.27.3 Member Function Documentation

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

Error handling.

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.

Implements murasaki::SpiSlaveStrategy.

12.27.3.2 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]

Data transfer to/from SPI slave.

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.

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

Callback to notify the end of transfer.

Parameters

ptr	Pointer to the control object.

Returns

true if no error.

Implements murasaki::SpiSlaveStrategy.

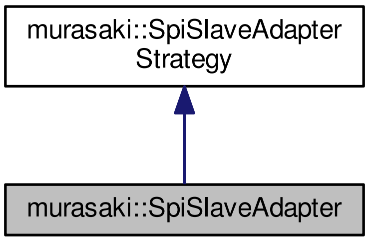
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spislave.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/spislave.cpp

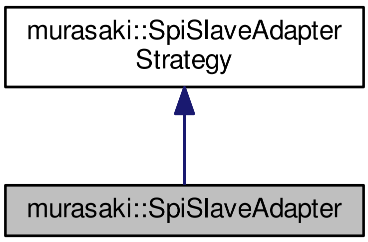
12.28 murasaki::SpiSlaveAdapter Class Reference

`#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)
- SpiSlaveAdapter (unsigned int pol, unsigned int pha,::GPIO_TypeDef ∗const port, uint16_t pin)
- virtual void AssertCs ()
- virtual void DeassertCs ()

12.28.1 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.

12.28.2 Constructor & Destructor Documentation

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

Constructor.

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.

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

Constructor.

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.

12.28.3 Member Function Documentation

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

Chip select assertion.

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

Reimplemented from murasaki::SpiSlaveAdapterStrategy.

12.28.3.2 void murasaki::SpiSlaveAdapter::DeassertCs ( ) [virtual]

Chip select deassertoin.

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

Reimplemented from murasaki::SpiSlaveAdapterStrategy.

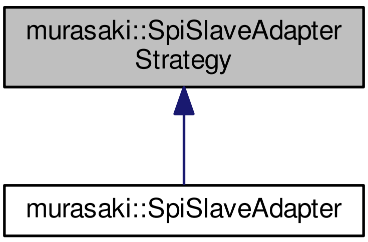
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spislaveadapter.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/spislaveadapter.cpp

12.29 murasaki::SpiSlaveAdapterStrategy Class Reference

`#include <spislaveadapterstrategy.hpp>`

Inheritance diagram for murasaki::SpiSlaveAdapterStrategy:



Public Member Functions

- SpiSlaveAdapterStrategy (murasaki::SpiClockPolarity pol, murasaki::SpiClockPhase pha)
- SpiSlaveAdapterStrategy (unsigned int pol, unsigned int pha)
- virtual void AssertCs ()
- virtual void DeassertCs ()
- murasaki::SpiClockPhase GetCpha ()
- murasaki::SpiClockPolarity GetCpol ()

12.29.1 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.

12.29.2 Constructor & Destructor Documentation

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

Constructor.

Parameters

pol	Polarity setting
pha	Phase setting

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

Constructor.

Parameters

pol	Polarity setting
pha	Phase setting

12.29.3 Member Function Documentation

12.29.3.1 void murasaki::SpiSlaveAdapterStrategy::AssertCs ( ) [virtual]

Chip select assertion.

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

This have to be overriden.

Reimplemented in murasaki::SpiSlaveAdapter.

12.29.3.2 void murasaki::SpiSlaveAdapterStrategy::DeassertCs ( ) [virtual]

Chip select deassertoin.

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

This have to be overriden.

Reimplemented in murasaki::SpiSlaveAdapter.

12.29.3.3 murasaki::SpiClockPhase murasaki::SpiSlaveAdapterStrategy::GetCpha ( )

Getter of the CPHA.

Returns

CPHA setting

12.29.3.4 murasaki::SpiClockPolarity murasaki::SpiSlaveAdapterStrategy::GetCpol ( )

Getter of the CPOL.

Returns

CPOL setting

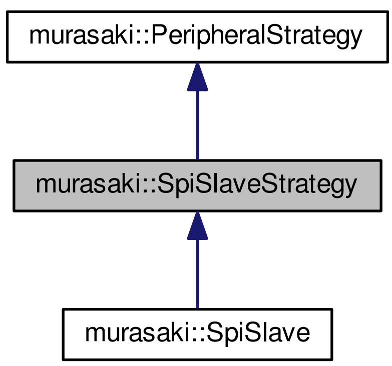
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spislaveadapterstrategy.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/spislaveadapterstrategy.cpp

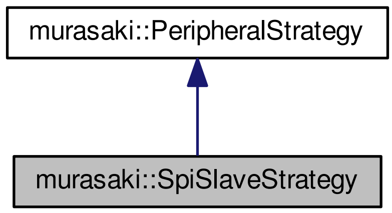
12.30 murasaki::SpiSlaveStrategy Class Reference

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

Additional Inherited Members

12.30.1 Detailed Description

This class provides a thread safe, synchronous SPI transfer.

12.30.2 Member Function Documentation

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

Handling error report of device.

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.

12.30.2.2 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]

Thread safe, synchronous SPI transfer.

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.

12.30.2.3 virtual bool murasaki::SpiSlaveStrategy::TransmitAndReceiveCompleteCallback (void * *ptr*) [pure virtual]

Callback to notifiy the end of transfer.

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:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spislavestrategy.hpp

12.31 murasaki::Synchronizer Class Reference

```
#include <synchronizer.hpp>
```

Public Member Functions

- bool Wait (unsigned int timeout_ms=kwmsIndefinitely)
- void Release ()

12.31.1 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.

12.31.2 Member Function Documentation

12.31.2.1 void murasaki::Synchronizer::Release ( )

Release the task.

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

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

Let the task wait for an interrupt.

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.

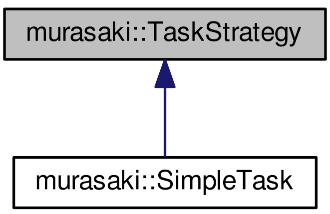
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/synchronizer.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/synchronizer.cpp

12.32 murasaki::TaskStrategy Class Reference

```
#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)
- void Start ()
- const char ∗ GetName ()
- unsigned int getStackDepth ()
- int getStackMinHeadroom ()

Protected Member Functions

- virtual void TaskBody (const void ∗ptr)=0

Static Protected Member Functions

- static void Launch (void ∗ptr)

12.32.1 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/`

12.32.2 Constructor & Destructor Documentation

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

Contractor. Task entity is not created here.

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.

12.32.3 Member Function Documentation

12.32.3.1 const char * murasaki::TaskStrategy::GetName ( )

Get a name of task.

Returns

A name of task.

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

Obtain the size of the stack.

Returns

Total depth of the task stack [byte]

12.32.3.3 int murasaki::TaskStrategy::getStackMinHeadroom ( )

Obtain the headroom of the stack.

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.

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

Internal use only. Create a task from TaskBody()

Parameters

ptr	passing "this" pointer.

12.32.3.5 void murasaki::TaskStrategy::Start (void)

Create a task and run it.

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

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

Actual task entity. Must be overridden by programmer.

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.

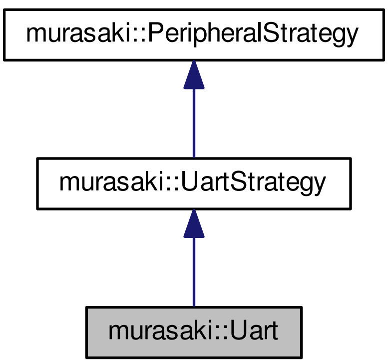
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/taskstrategy.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/taskstrategy.cpp

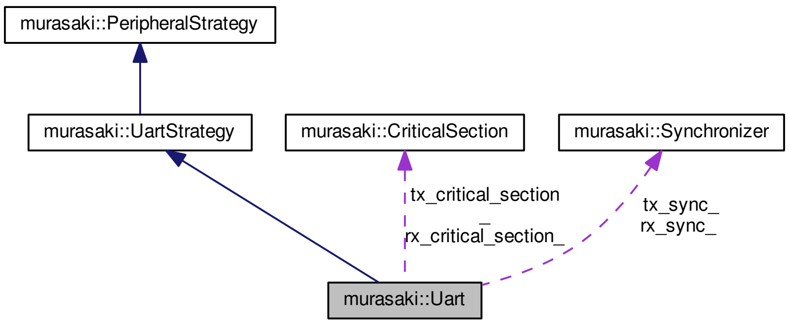
12.33 murasaki::Uart Class Reference

```
#include <uart.hpp>
```

Inheritance diagram for murasaki::Uart:



Collaboration diagram for murasaki::Uart:



Public Member Functions

- Uart (UART_HandleTypeDef ∗uart)
- virtual void SetHardwareFlowControl (UartHardwareFlowControl control)
- virtual void SetSpeed (unsigned int baud_rate)
- virtual murasaki::UartStatus Transmit (const uint8_t ∗data, unsigned int size, unsigned int timeout_ms)
- virtual murasaki::UartStatus Receive (uint8_t ∗data, unsigned int count, unsigned int ∗transfered_count, UartTimeout uart_timeout, unsigned int timeout_ms)
- virtual bool TransmitCompleteCallback (void ∗const ptr)
- virtual bool ReceiveCompleteCallback (void ∗const ptr)
- virtual bool HandleError (void ∗const ptr)

Additional Inherited Members

12.33.1 Detailed Description

The Uart class is the wrapper of the UART controller. To use the Uart class, make an instance with UART_Handle↩ TypeDef ∗ 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.

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::TransmitCompleteCallback() 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 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.

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.

12.33.2 Constructor & Destructor Documentation

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

Constructor.

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.

12.33.3 Member Function Documentation

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

Error handling.

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.

Implements murasaki::UartStrategy.

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

Receive raw data through an UART by synchronous mode.

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.

12.33.3.3 bool murasaki::Uart::ReceiveCompleteCallback (void *const *ptr*) [virtual]

Call back for entire block transfer completion.

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.

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

Set the behavior of the hardware flow control.

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.

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

Set the BAUD rate.

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.

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

Transmit raw data through an UART by synchronous mode.

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.

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

Call back for entire block transfer completion.

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

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.

Implements murasaki::UartStrategy.

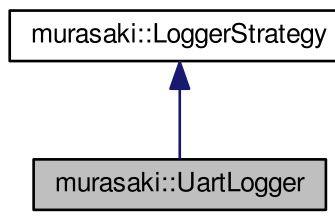
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/uart.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/uart.cpp

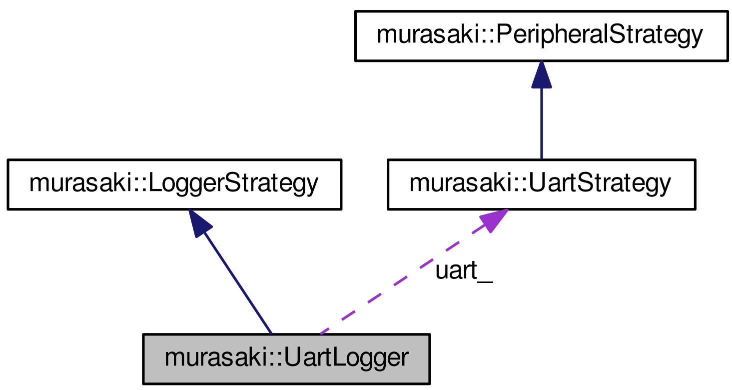
12.34 murasaki::UartLogger Class Reference

```
#include <uartlogger.hpp>
```

Inheritance diagram for murasaki::UartLogger:



Collaboration diagram for murasaki::UartLogger:



Public Member Functions

- UartLogger (UartStrategy ∗uart)
- virtual void putMessage (char message[ ], unsigned int size)
- virtual char getCharacter ()
- virtual void DoPostMortem (void ∗debugger_fifo)

12.34.1 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.

12.34.2 Constructor & Destructor Documentation

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

Constructor.

Parameters

uart	Pointer to the uart object.

12.34.3 Member Function Documentation

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

Start post mortem process.

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.

12.34.3.2 char murasaki::UartLogger::getCharacter ( ) [virtual]

Character input member function.

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.

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

Message output member function.

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.

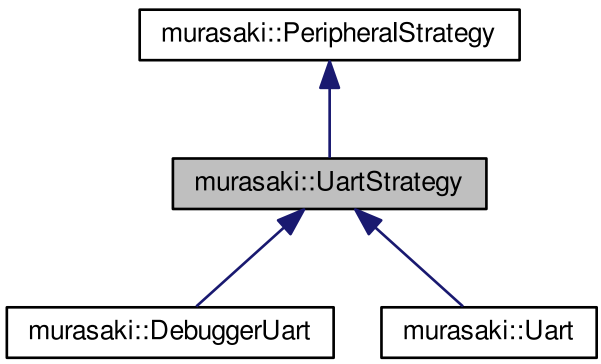
The documentation for this class was generated from the following files:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/uartlogger.hpp
- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/uartlogger.cpp

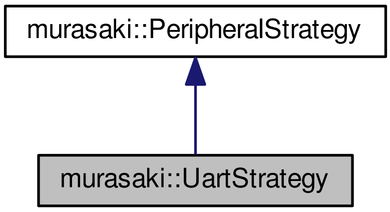
12.35 murasaki::UartStrategy Class Reference

```
#include <uartstrategy.hpp>
```

Inheritance diagram for murasaki::UartStrategy:



Collaboration diagram for murasaki::UartStrategy:



Public Member Functions

- virtual void SetHardwareFlowControl (UartHardwareFlowControl control)
- virtual void SetSpeed (unsigned int speed)
- virtual murasaki::UartStatus Transmit (const uint8_t *data, unsigned int size, unsigned int timeout_ms=murasaki::kwmsIndefinitely)=0
- 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
- virtual bool TransmitCompleteCallback (void *ptr)=0
- virtual bool ReceiveCompleteCallback (void *ptr)=0
- virtual bool HandleError (void *ptr)=0

Additional Inherited Members

12.35.1 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.

12.35.2 Member Function Documentation

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

Handling error report of device.

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.

12.35.2.2 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]`

buffer receive over the UART. synchronous

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.

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

Call back to be called for entire block transfer is complete.

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.

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

Set the behavior of the hardware flow control.

Parameters

control	The control mode.

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

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

the baud rate

Parameters

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

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

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

buffer transmission over the UART. synchronous

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::DebuggerUart, and murasaki::Uart.

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

Call back to be called notify the transfer is complete.

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:

- /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/uartstrategy.hpp

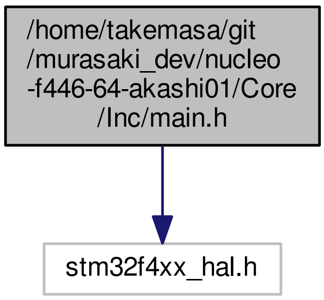
Chapter 13

File Documentation

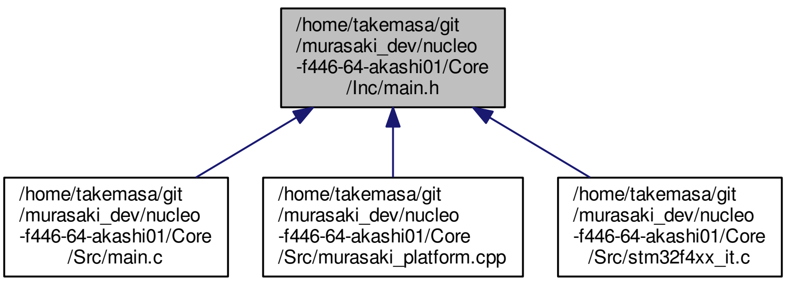
13.1 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Inc/main.h File Reference

```
#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)

13.1.1 Detailed Description

Attention

13.1.2 Function Documentation

13.1.2.1 void Error_Handler (void)

This function is executed in case of error occurrence.

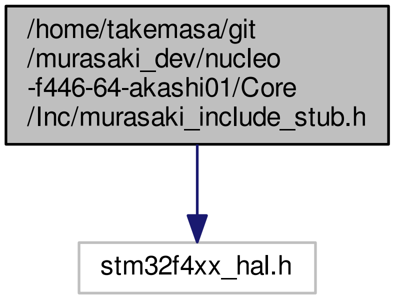
Return values

None	

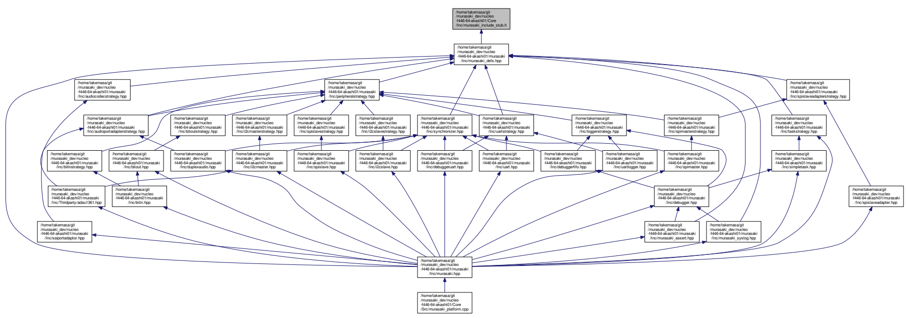
13.2 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Inc/murasaki_include_stub.h File Reference

```
#include <stm32f4xx_hal.h>
```

Include dependency graph for murasaki_include_stub.h:



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



13.2.1 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"
```

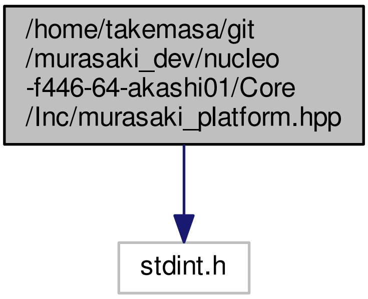
This is easy to understand. But boring to maintain.

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.

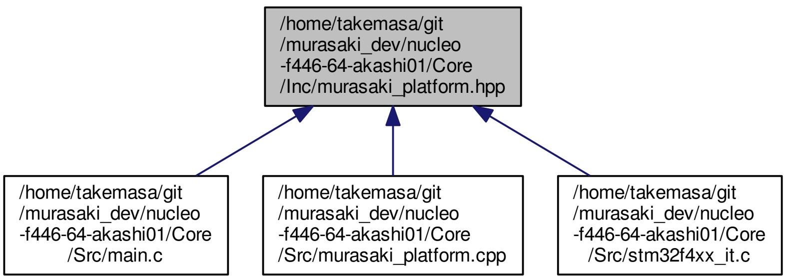
13.3 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Inc/murasaki_platform.hpp File Reference

`#include <stdint.h>`

Include dependency graph for murasaki_platform.hpp:



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



Functions

- void InitPlatform ()
- void ExecPlatform ()
- void CustomAssertFailed (uint8_t ∗file, uint32_t line)
- void CustomDefaultHandler ()
- void PrintFaultResult (unsigned int ∗stack_pointer)
- void MasterTaskBodyFunction (const void ∗ptr)
- void SlaveTaskBodyFunction (const void ∗ptr)

13.3.1 Detailed Description

Date

2017/11/12

Author

Seiichi "Suikan" Horie

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

13.3.2 Function Documentation

13.3.2.1 void MasterTaskBodyFunction (const void ∗ *ptr*)

Master test task.

Parameters

ptr	Pointer to the parameter block

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

13.3.2.2 void PrintFaultResult (unsigned int ∗ *stack_pointer*)

Printing out the context information.

Parameters

stack_pointer	retrieved stack pointer before interrupt / exception.

Do not call from application. This is murasaki_internal_only.

13.3.2.3 void SlaveTaskBodyFunction (const void ∗ *ptr*)

Demonstration task.

Parameters

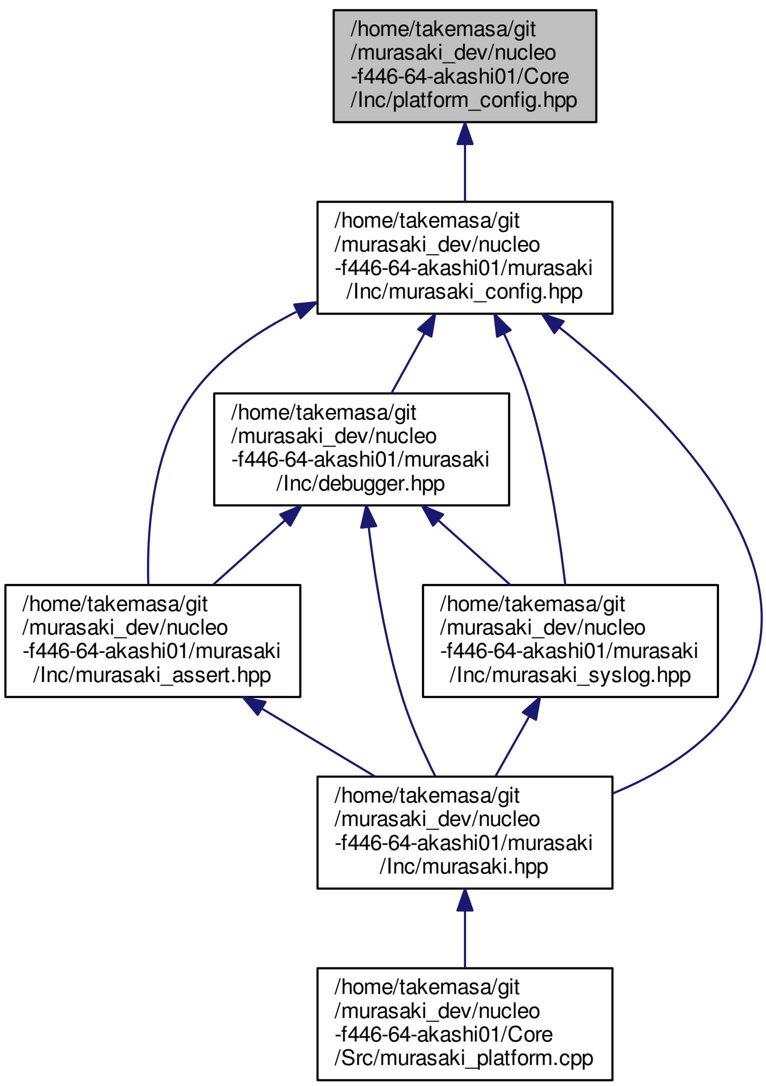
ptr	Pointer to the parameter block

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

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

13.4 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Inc/platform_↩ config.hpp File Reference

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



Macros

- #define MURASAKI_CONFIG_NOSYSLOG false

13.4.1 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.

13.4.2 Macro Definition Documentation

13.4.2.1 #define MURASAKI_CONFIG_NOSYSLOG false

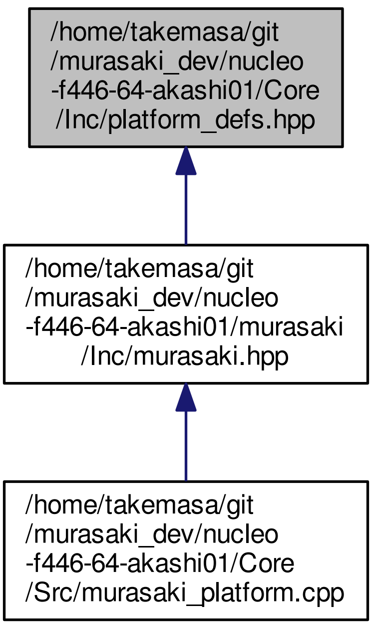
Suppress MURASAKI_SYSLOG macro.

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.

13.5 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Inc/platform_defs.hpp File Reference

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



Classes

- struct murasaki::Platform

Namespaces

- murasaki

Variables

- Platform murasaki::platform

13.5.1 Detailed Description

Date

2018/01/16

Author

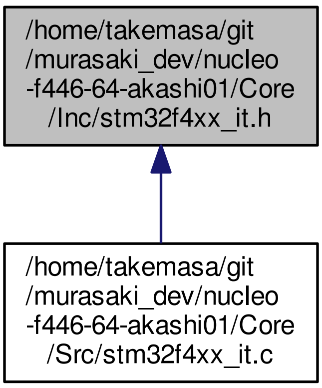
Seiichi "Suikan" Horie

This file contains user defined struct murasaki::Platform.

This file will be included by murasaki.hpp.

13.6 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Inc/stm32f4xx_it.h File Reference

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



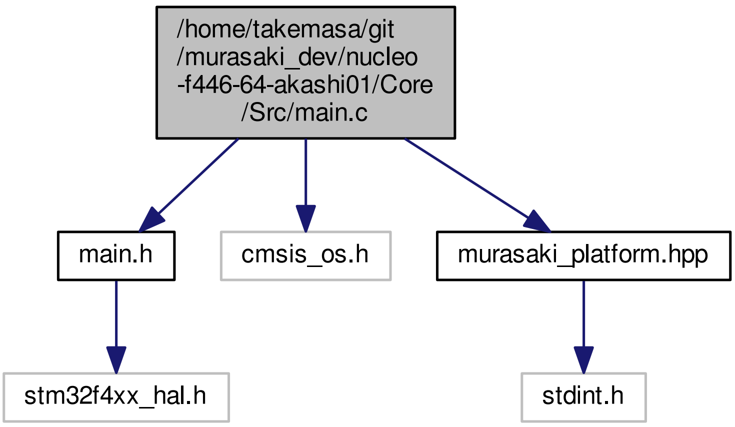
13.6.1 Detailed Description

Attention

13.7 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Src/main.c File Reference

```
#include "main.h"
#include "cmsis_os.h"
#include "murasaki_platform.hpp"
```

Include dependency graph for main.c:



Functions

- void SystemClock_Config (void)
- static void MX_GPIO_Init (void)
- static void MX_DMA_Init (void)
- static void MX_USART2_UART_Init (void)
- static void MX_I2C1_Init (void)

- static void MX_I2C2_Init (void)
- static void MX_SPI1_Init (void)
- static void MX_SPI2_Init (void)
- static void MX_USART1_UART_Init (void)
- static void MX_TIM2_Init (void)
- void StartDefaultTask (void const ∗argument)
- int main (void)
- void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef ∗htim)
- void Error_Handler (void)
- void assert_failed (uint8_t ∗file, uint32_t line)

Variables

- DMA_HandleTypeDef hdma_spi1_rx

13.7.1 Detailed Description

Attention

13.7.2 Function Documentation

13.7.2.1 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.

Parameters

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

Return values

None	

13.7.2.2 void Error_Handler (void)

This function is executed in case of error occurrence.

Return values

None	

13.7.2.3 void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef * *htim*)

Period elapsed callback in non blocking mode.

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	

13.7.2.4 int main (void)

The application entry point.

Return values

int	

13.7.2.5 static void MX_DMA_Init (void) [static]

Enable DMA controller clock

13.7.2.6 static void MX_GPIO_Init (void) [static]

GPIO Initialization Function.

Parameters

None	

Return values

None	

13.7.2.7 static void MX_I2C1_Init (void) [static]

I2C1 Initialization Function.

Parameters

None	

Return values

None	

13.7.2.8 static void MX_I2C2_Init (void) [static]

I2C2 Initialization Function.

Parameters

None	

Return values

None	

13.7.2.9 static void MX_SPI1_Init (void) [static]

SPI1 Initialization Function.

Parameters

None	

Return values

None	

13.7.2.10 static void MX_SPI2_Init (void) [static]

SPI2 Initialization Function.

Parameters

None	

Return values

None	

13.7.2.11 static void MX_TIM2_Init (void) [static]

TIM2 Initialization Function.

Parameters

None	

Return values

None	

13.7.2.12 static void MX_USART1_UART_Init (void) [static]

USART1 Initialization Function.

Parameters

None	

Return values

None	

13.7.2.13 static void MX_USART2_UART_Init (void) [static]

USART2 Initialization Function.

Parameters

None	

Return values

None	

13.7.2.14 void StartDefaultTask (void const * *argument*)

Function implementing the defaultTask thread.

Parameters

argument	Not used

Return values

None	

13.7.2.15 void SystemClock_Config (void)

System Clock Configuration.

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

13.7.3 Variable Documentation

13.7.3.1 DMA_HandleTypeDef hdma_spi1_rx

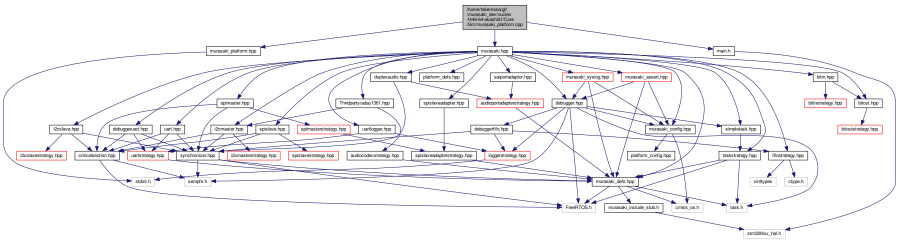
File Name : stm32f4xx_hal_msp.c Description : This file provides code for the MSP Initialization and de-Initialization codes.

Attention

13.8 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Src/murasaki_platform.cpp File Reference

```
#include <murasaki_platform.hpp>
#include "main.h"
#include "murasaki.hpp"
```

Include dependency graph for murasaki_platform.cpp:



Functions

- void I2cSearch (murasaki::I2CMasterStrategy ∗master)
- void InitPlatform ()
- void ExecPlatform ()
- 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_GPIO_EXTI_Callback (uint16_t GPIO_Pin)
- void CustomAssertFailed (uint8_t ∗file, uint32_t line)
- void PrintFaultResult (unsigned int ∗stack_pointer)

13.8.1 Detailed Description

Date

2018/05/20

Author

Seiichi "Suikan" Horie

13.8.2 Function Documentation

13.8.2.1 void HAL_I2C_MasterRxCpltCallback (I2C_HandleTypeDef ∗ *hi2c*)

Essential to sync up with I2C.

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.

13.8.2.2 void HAL_I2C_SlaveRxCpltCallback (I2C_HandleTypeDef * *hi2c*)

Essential to sync up with I2C.

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.

13.8.2.3 void I2cSearch (murasaki::I2CMasterStrategy * *master*)

I2C device serach function.

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.

This function can be deleted if you don't use.

13.8.2.4 void PrintFaultResult (unsigned int * *stack_pointer*)

Printing out the context information.

Parameters

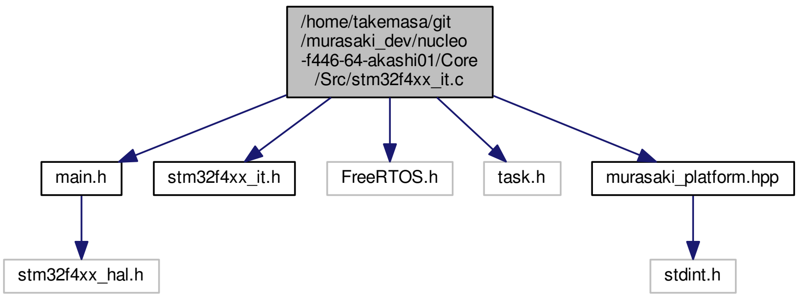
stack_pointer	retrieved stack pointer before interrupt / exception.

Do not call from application. This is murasaki_internal_only.

13.9 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Src/stm32f4xx_it.c File Reference

```
#include "main.h"
#include "stm32f4xx_it.h"
#include "FreeRTOS.h"
#include "task.h"
#include "murasaki_platform.hpp"
```

Include dependency graph for stm32f4xx_it.c:



Variables

- DMA_HandleTypeDef hdma_spi1_rx

13.9.1 Detailed Description

Attention

13.9.2 Variable Documentation

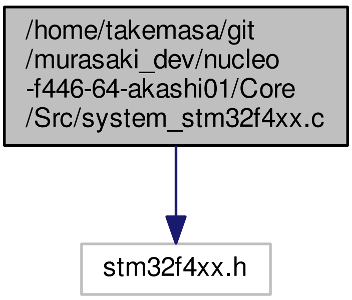
13.9.2.1 DMA_HandleTypeDef hdma_spi1_rx

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

Attention

13.10 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/Core/Src/system_stm32f4xx.c File Reference

```
#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)
- void SystemCoreClockUpdate (void)

13.10.1 Detailed Description

Author

MCD Application Team This file provides two functions and one global variable to be called from user application:

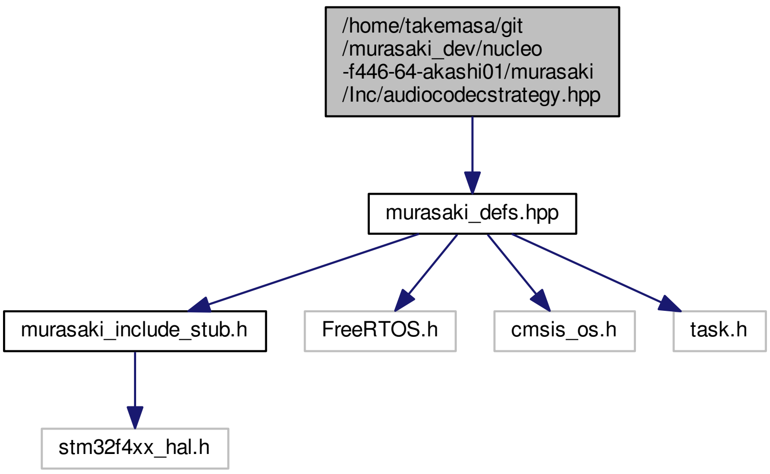
- SystemInit(): This function is called at startup just after reset and before branch to main program. This call is made inside the "startup_stm32f4xx.s" file.
- SystemCoreClock variable: Contains the core clock (HCLK), it can be used by the user application to setup the SysTick timer or configure other parameters.
- SystemCoreClockUpdate(): Updates the variable SystemCoreClock and must be called whenever the core clock is changed during program execution.

Attention

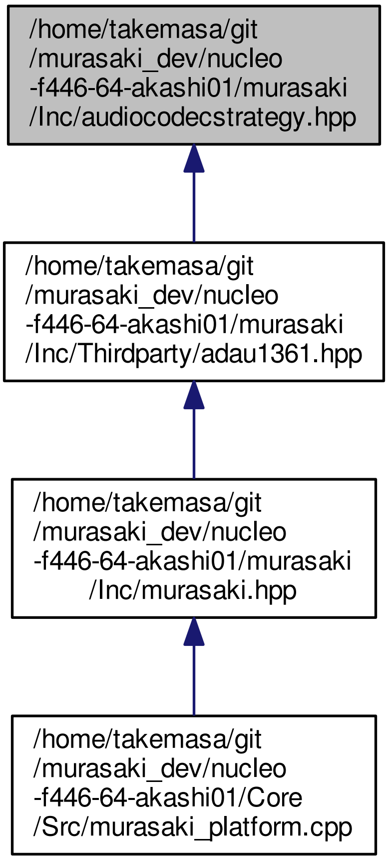
13.11 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/audiocodecstrategy.hpp File Reference

`#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

Namespaces

- murasaki

13.11.1 Detailed Description

Date

2018/05/11

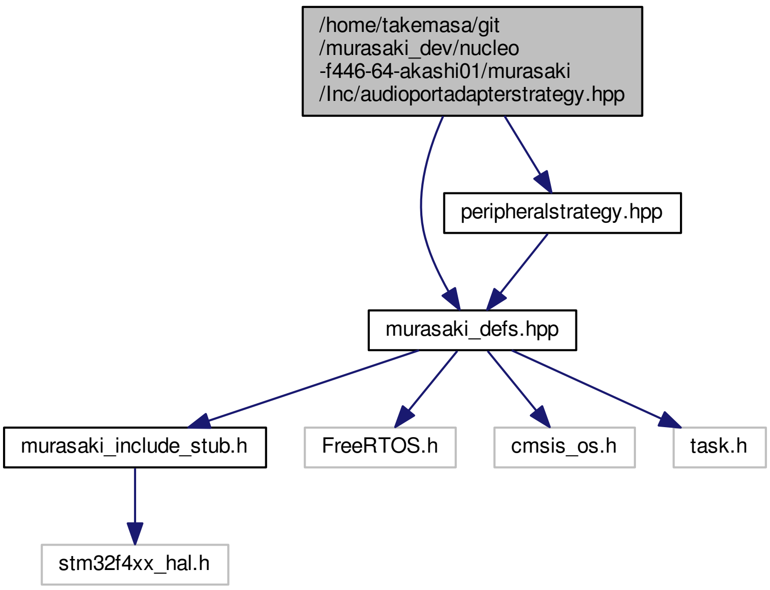
Author

: Seiichi "Suikan" Horie

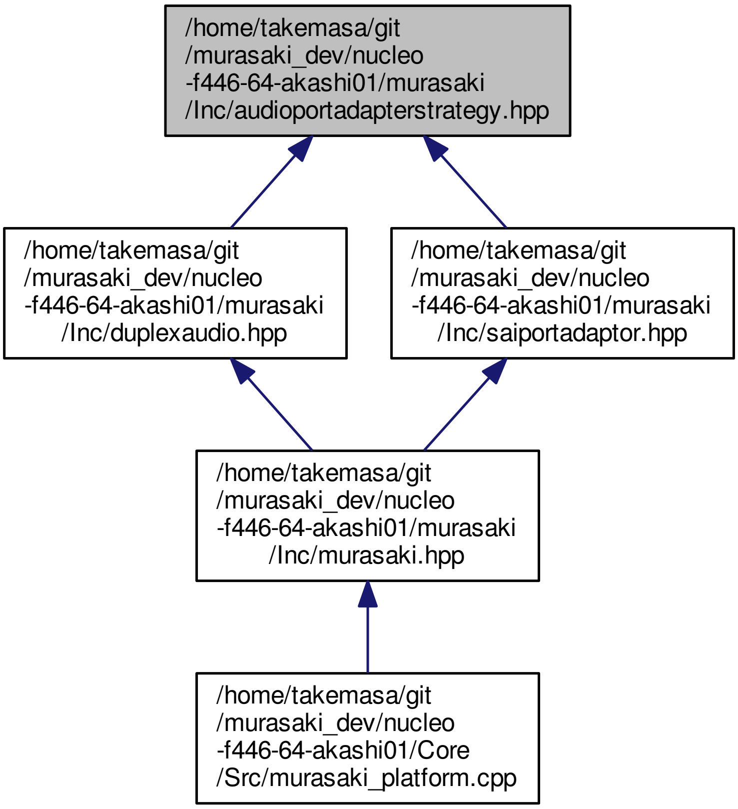
13.12 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/audioportadapterstrategy.hpp File Reference

```
#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

Namespaces

- murasaki

13.12.1 Detailed Description

Date

2019/07/28

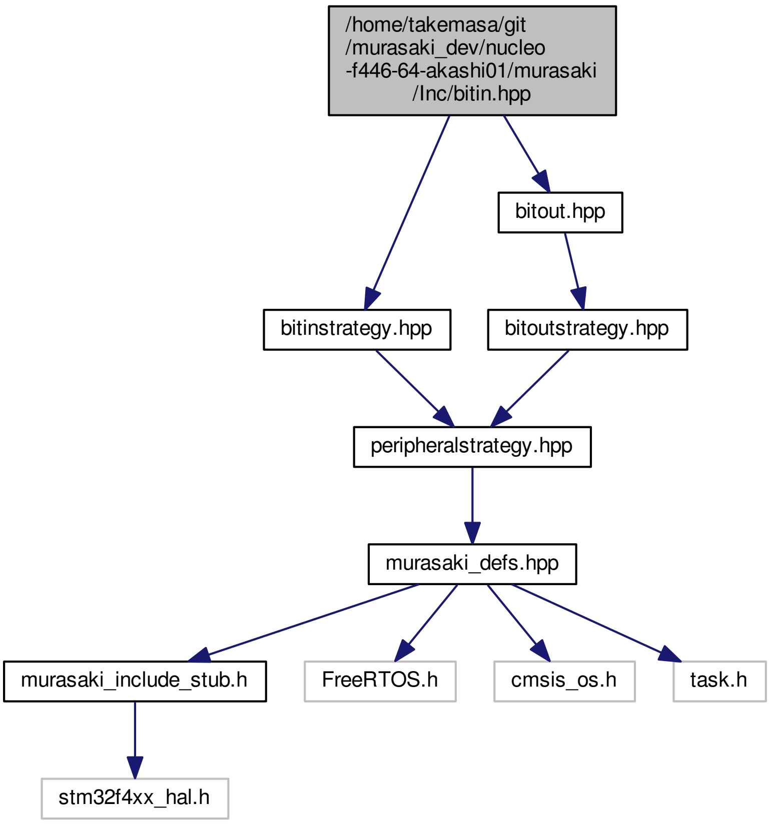
Author

Seiichi "Suikan" Horie

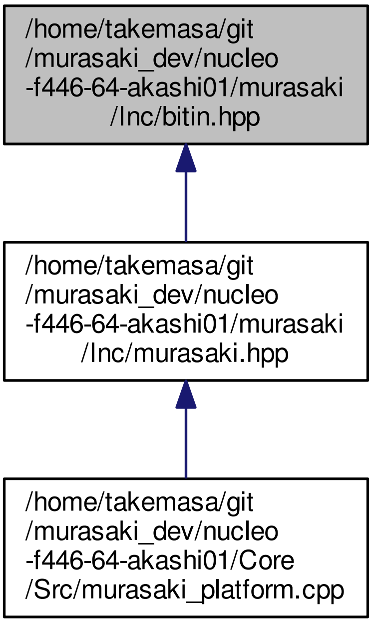
13.13 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitin.hpp File Reference

```
#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

Namespaces

- murasaki

13.13.1 Detailed Description

Date

2018/05/07

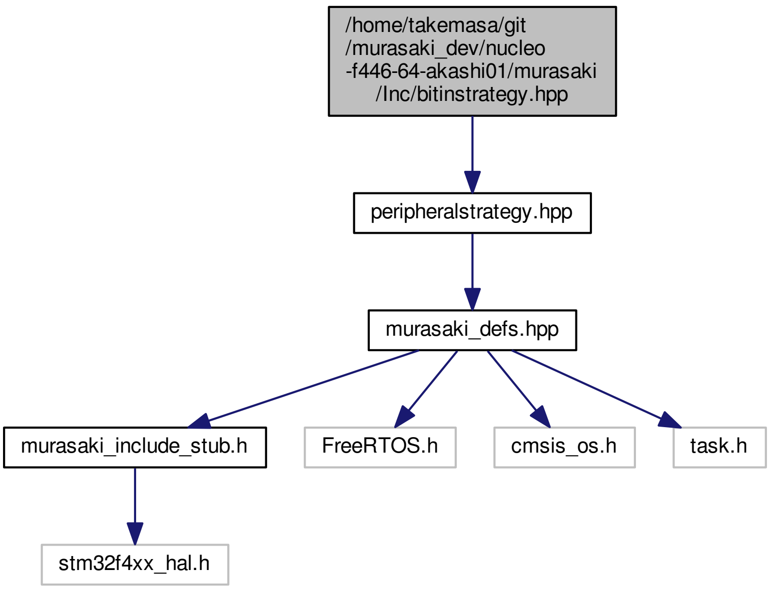
Author

Seiichi "Suikan" Horie

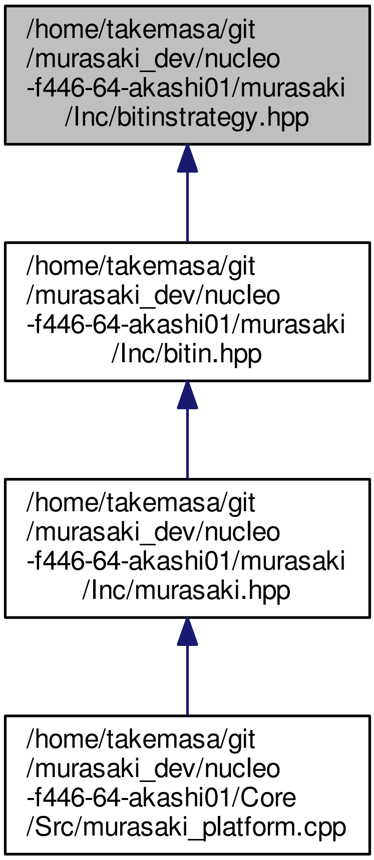
13.14 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitinstrategy.hpp File Reference

`#include <peripheralstrategy.hpp>`

Include dependency graph for bitinstrategy.hpp:



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



Classes

- class murasaki::BitInStrategy

Namespaces

- murasaki

13.14.1 Detailed Description

Date

2018/05/07

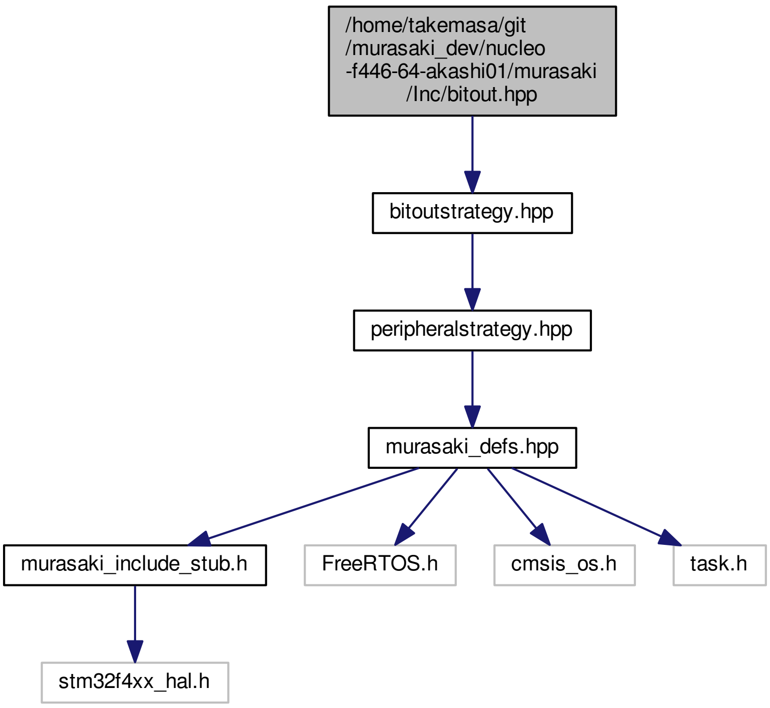
Author

Seiichi "Suikan" Horie

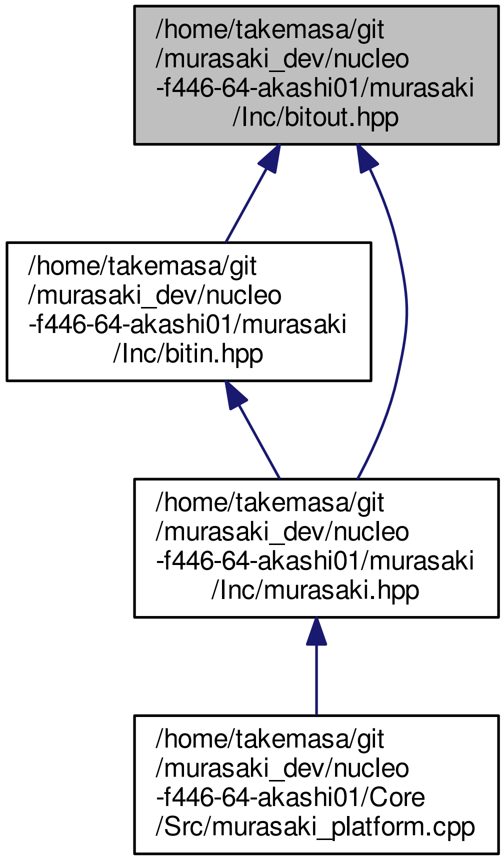
13.15 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitout.hpp File Reference

```
#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
- class murasaki::BitOut

Namespaces

- murasaki

13.15.1 Detailed Description

Date

2018/05/07

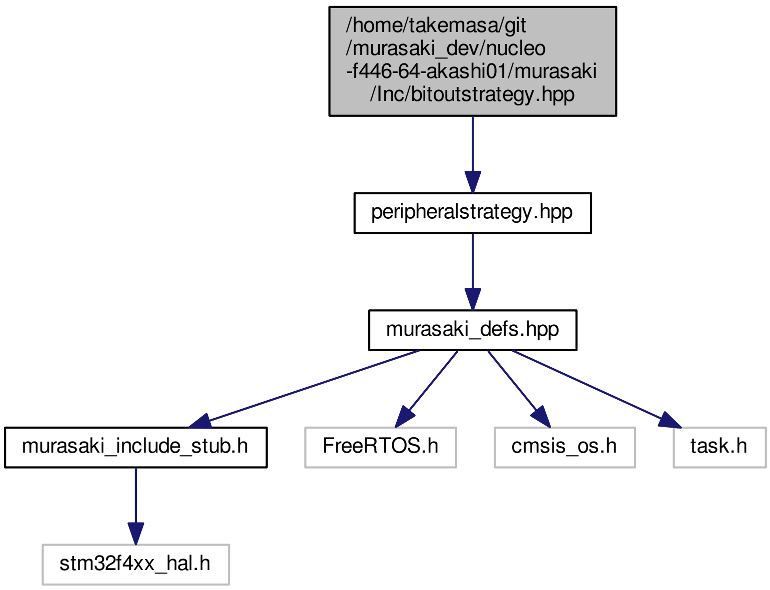
Author

Seiichi "Suikan" Horie

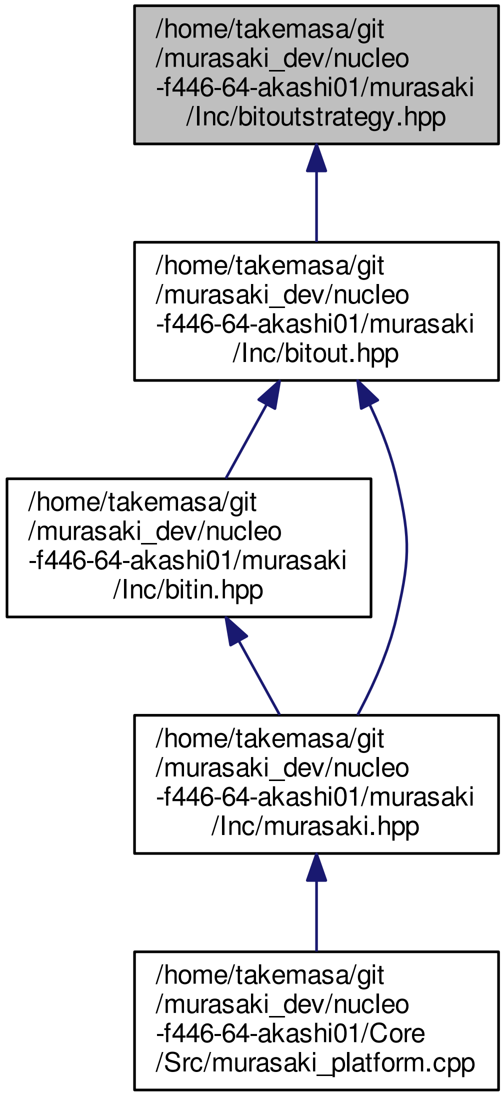
13.16 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/bitoutstrategy.hpp File Reference

```
#include <peripheralstrategy.hpp>
```

Include dependency graph for bitoutstrategy.hpp:



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



Classes

- class murasaki::BitOutStrategy

Namespaces

- murasaki

13.16.1 Detailed Description

Date

2018/05/07

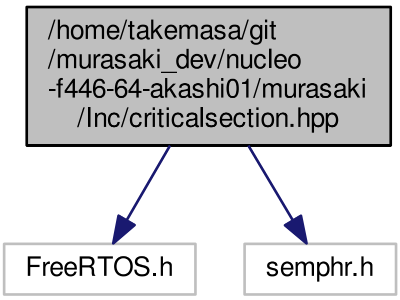
Author

Seiichi "Suikan" Horie

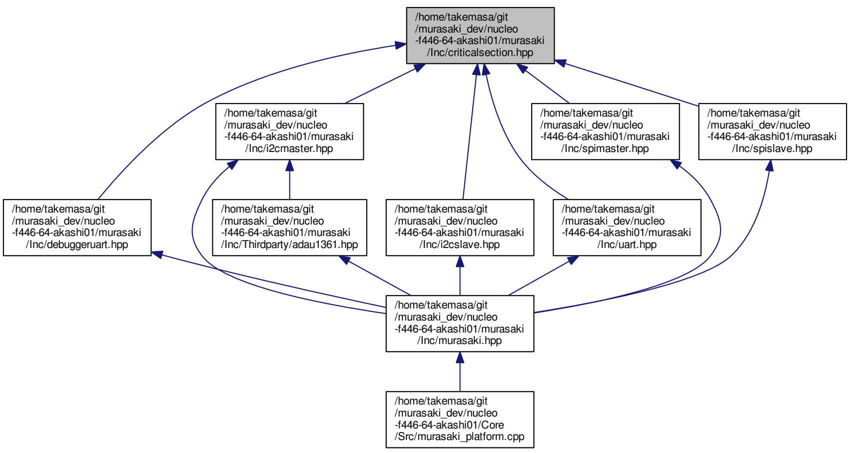
13.17 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/criticalsection.hpp File Reference

```
#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

Namespaces

- murasaki

13.17.1 Detailed Description

Date

2018/01/27

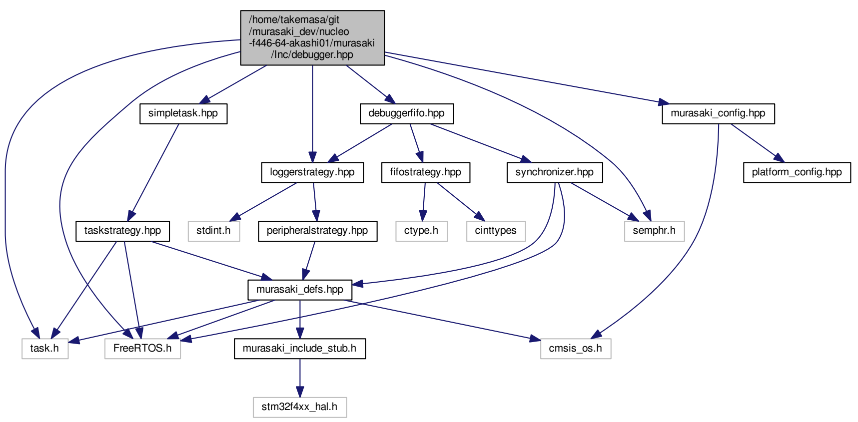
Author

Seiichi "Suikan" Horie

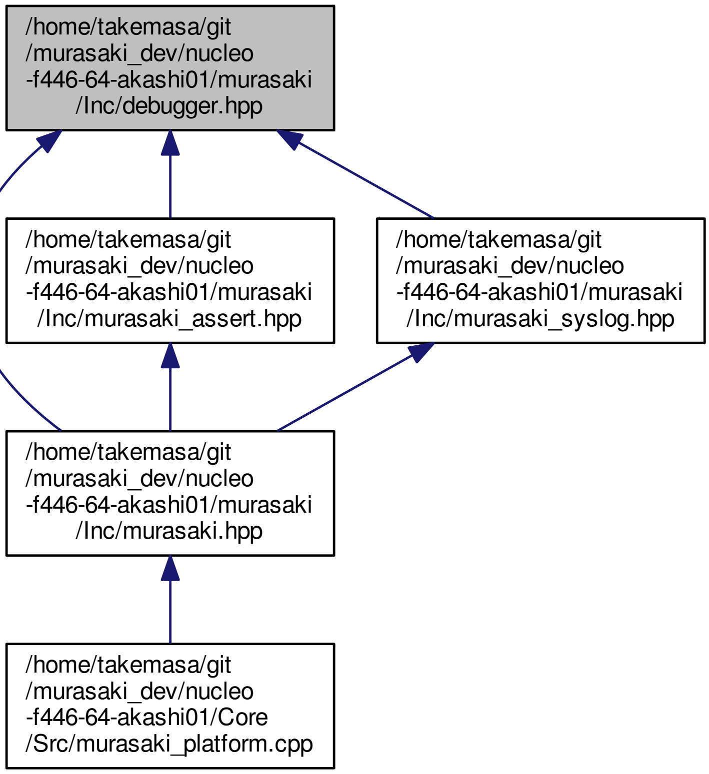
13.18 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/debugger.hpp File Reference

```
#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

Namespaces

- murasaki

Variables

- Debugger ∗ murasaki::debugger

13.18.1 Detailed Description

Date

2018/01/03

Author

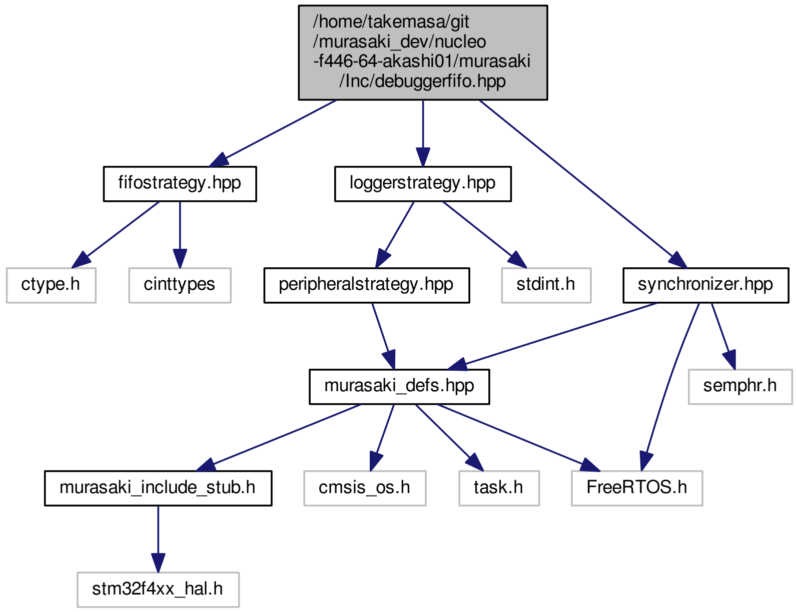
Seiichi "Suikan" Horie

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

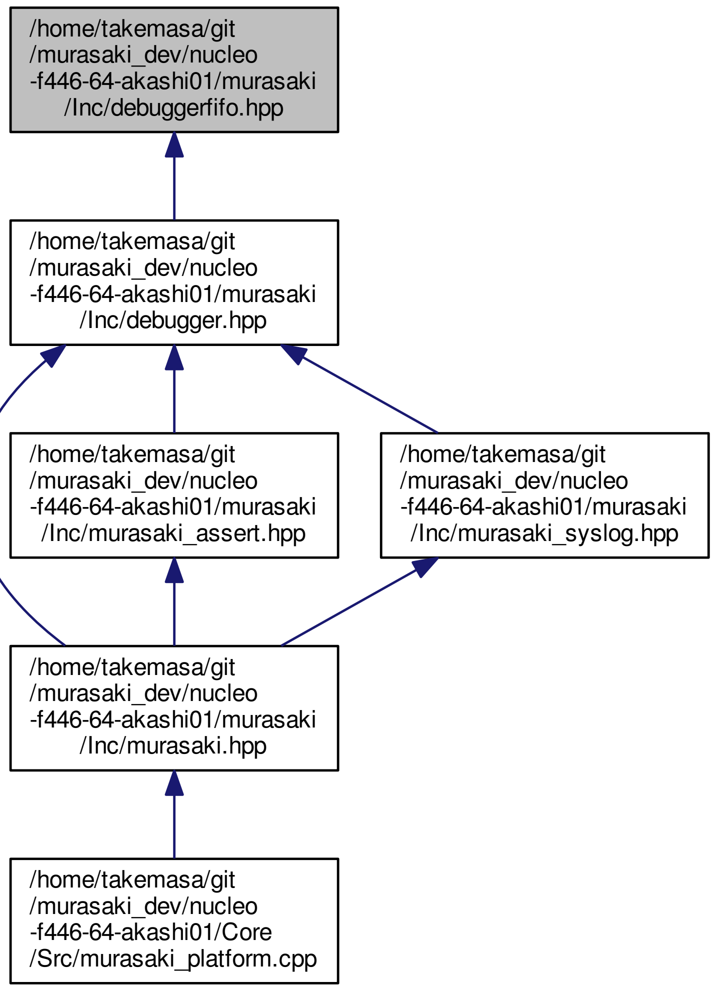
13.19 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/debuggerfifo.hpp File Reference

```
#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
- struct murasaki::LoggingHelpers

Namespaces

- murasaki

13.19.1 Detailed Description

Date

2018/03/01

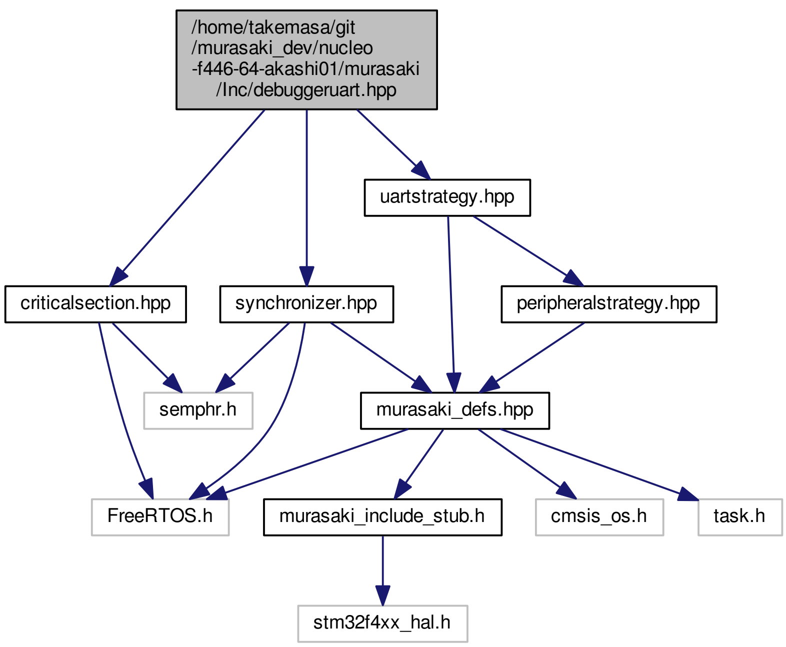
Author

Seiichi "Suikan" Horie

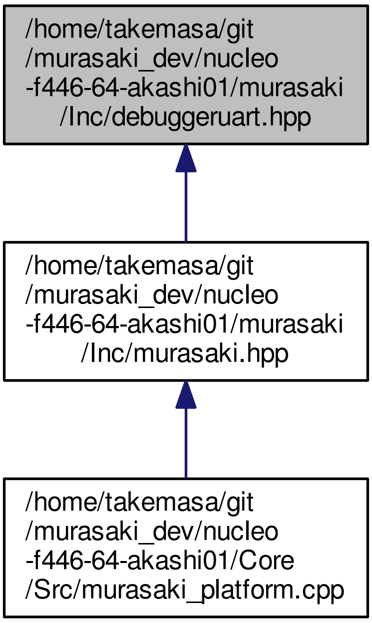
13.20 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/debuggeruart.hpp File Reference

```
#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

Namespaces

- murasaki

13.20.1 Detailed Description

Date

2018/09/23

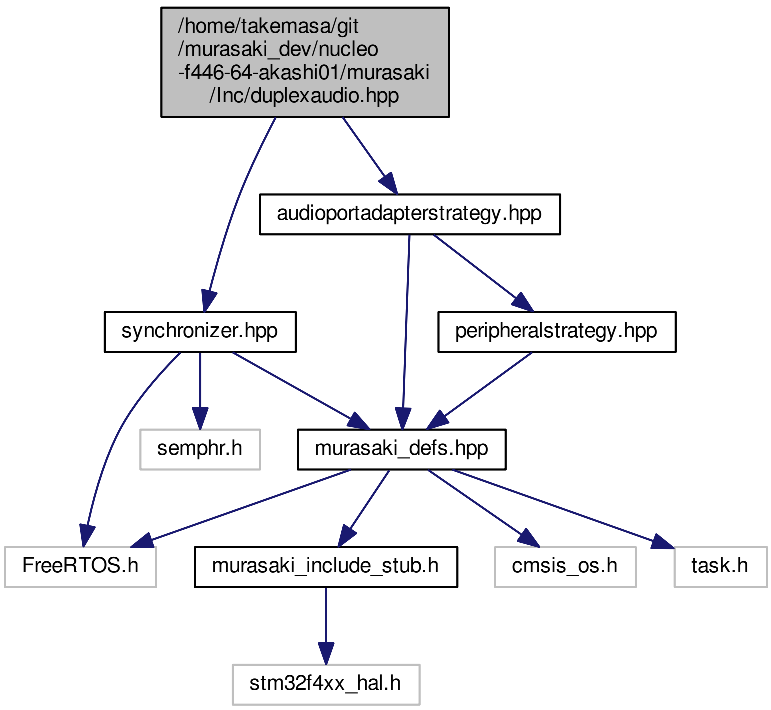
Author

Seiichi "Suikan" Horie

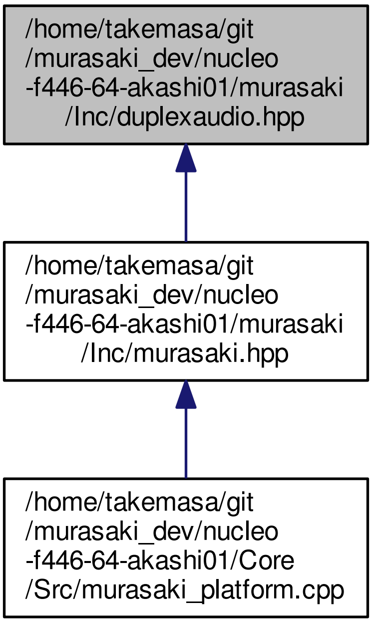
13.21 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/duplexaudio.hpp File Reference

```
#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

Namespaces

- murasaki

13.21.1 Detailed Description

Date

2019/03/02

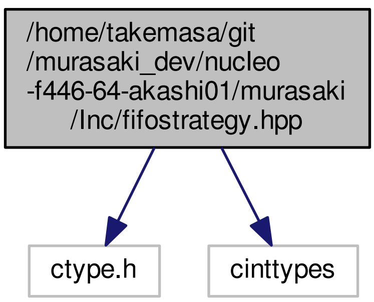
Author

Seiichi "Suikan" Horie

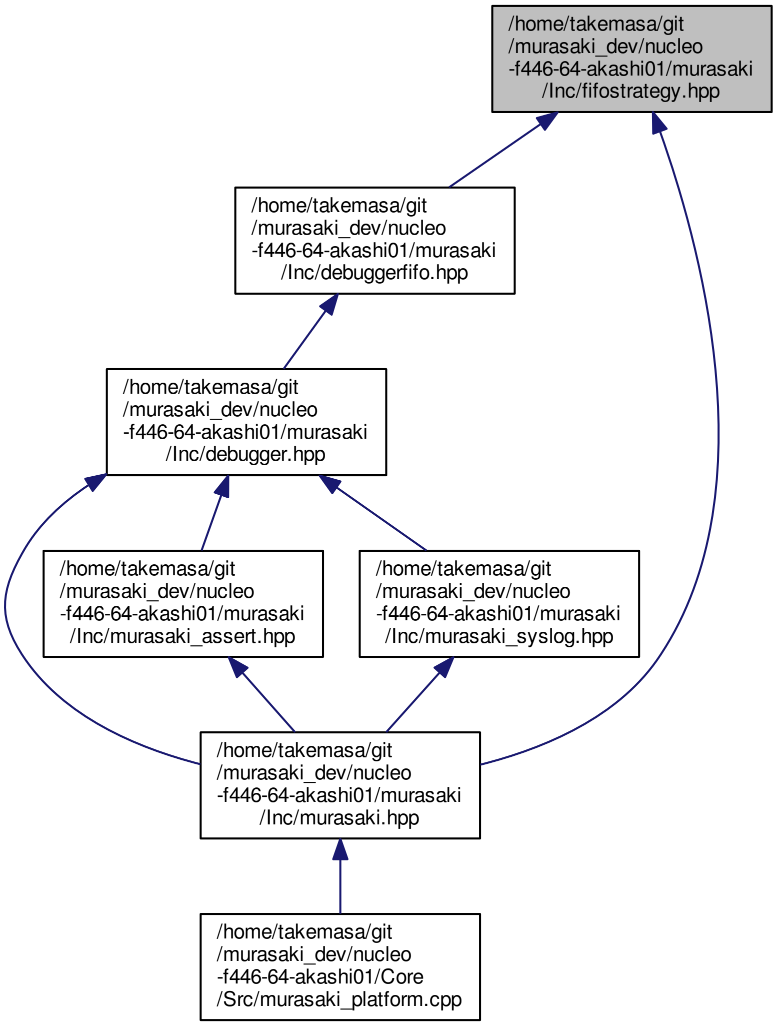
13.22 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/fifostrategy.hpp File Reference

```
#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

Namespaces

- murasaki

13.22.1 Detailed Description

Date

2018/02/26

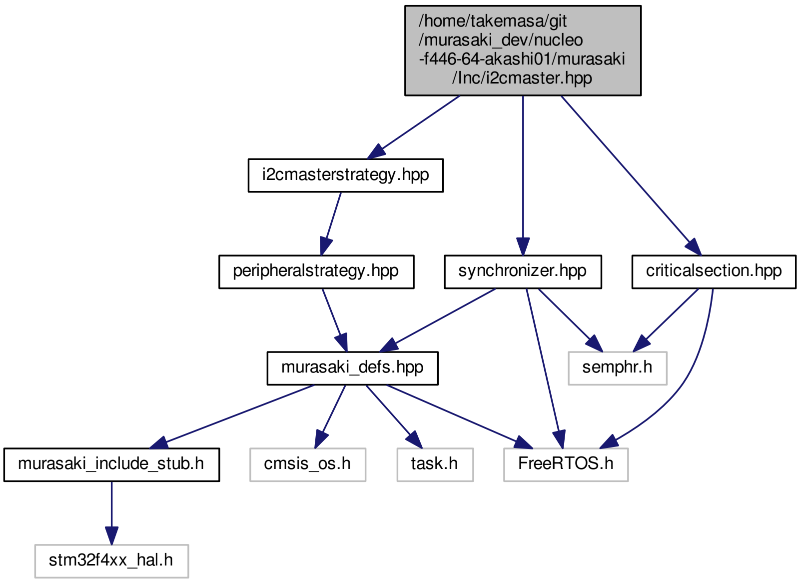
Author

Seiichi "Suikan" Horie

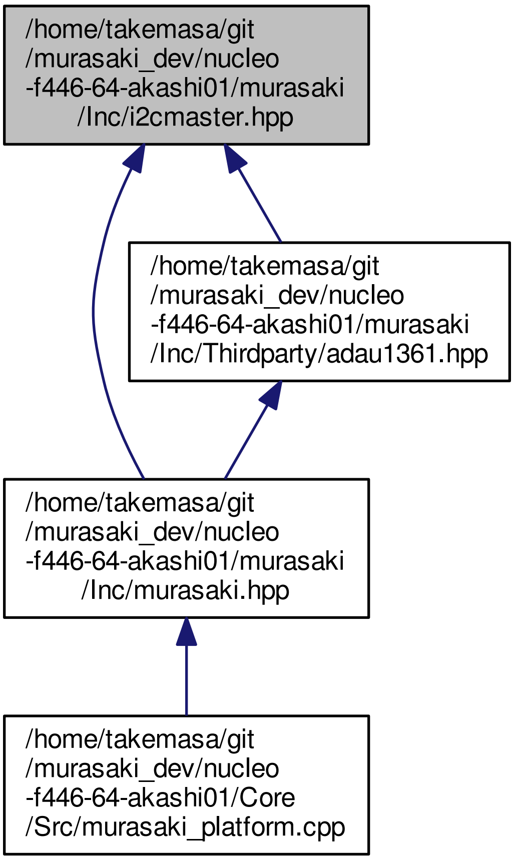
13.23 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/i2cmaster.hpp File Reference

```
#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

Namespaces

- murasaki

13.23.1 Detailed Description

Date

2018/02/12

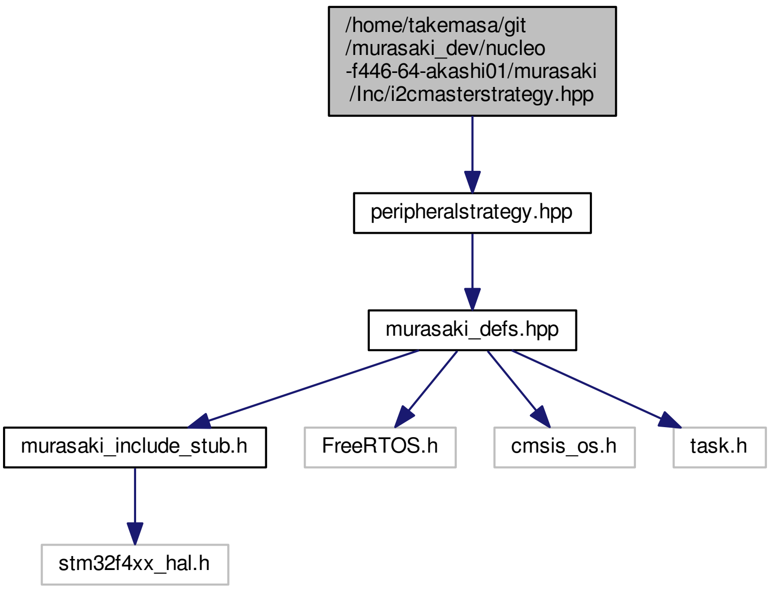
Author

: Seiichi "Suikan" Horie

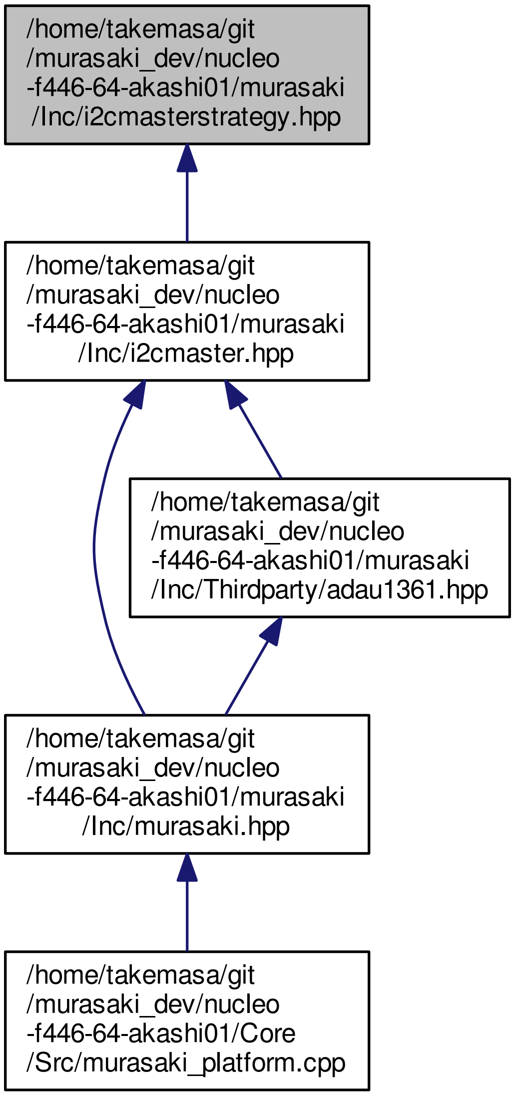
13.24 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/i2cmasterstrategy.hpp File Reference

`#include <peripheralstrategy.hpp>`

Include dependency graph for i2cmasterstrategy.hpp:



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



Classes

- class murasaki::I2CMasterStrategy

Namespaces

- murasaki

13.24.1 Detailed Description

Date

2018/02/11

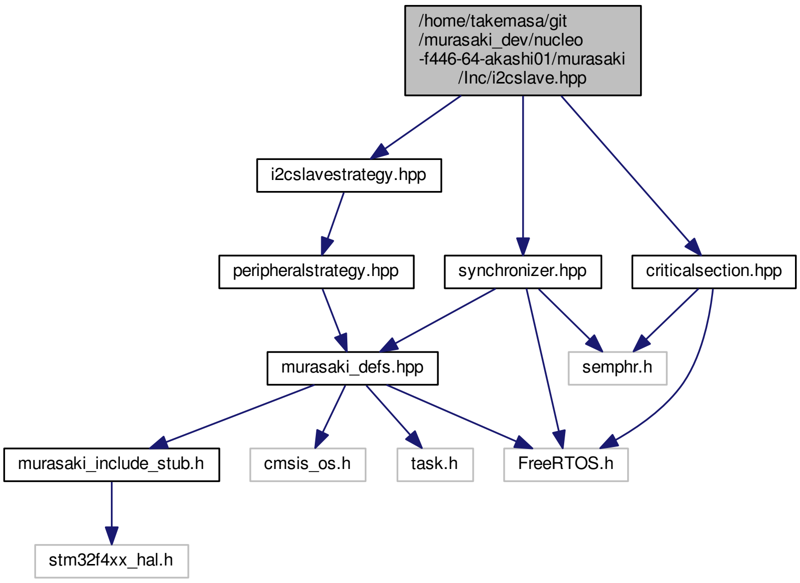
Author

: Seiichi "Suikan" Horie

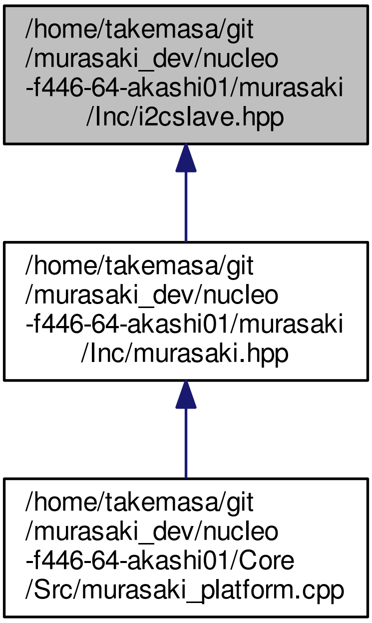
13.25 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/i2cslave.hpp File Reference

```
#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

Namespaces

- murasaki

13.25.1 Detailed Description

Date

2018/10/07

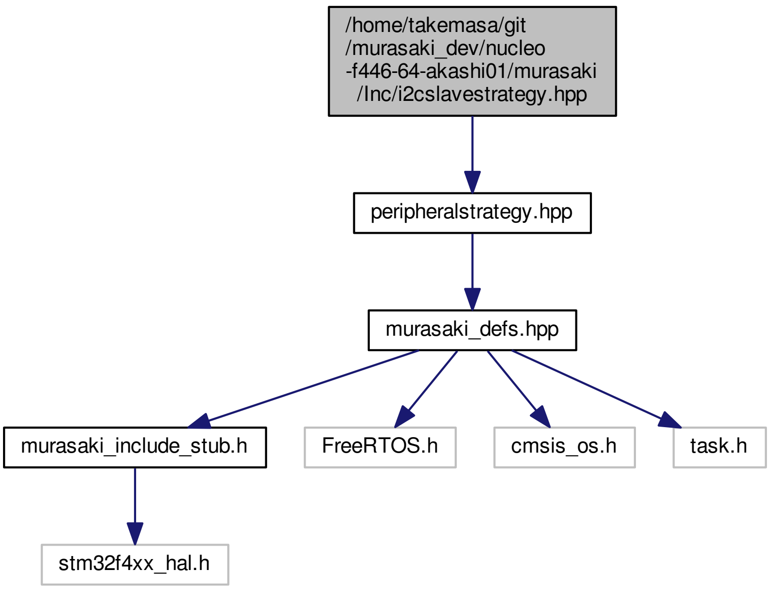
Author

Seiichi "Suikan" Horie

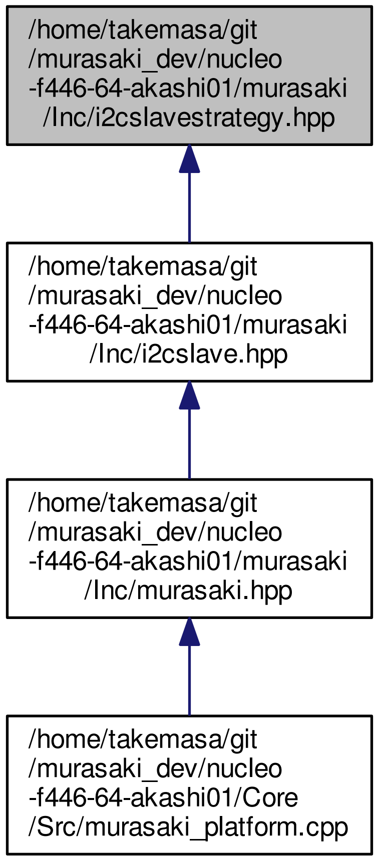
13.26 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/i2cslavestrategy.hpp File Reference

`#include <peripheralstrategy.hpp>`

Include dependency graph for i2cslavestrategy.hpp:



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



Classes

- class murasaki::I2cSlaveStrategy

Namespaces

- murasaki

13.26.1 Detailed Description

Date

2018/10/07

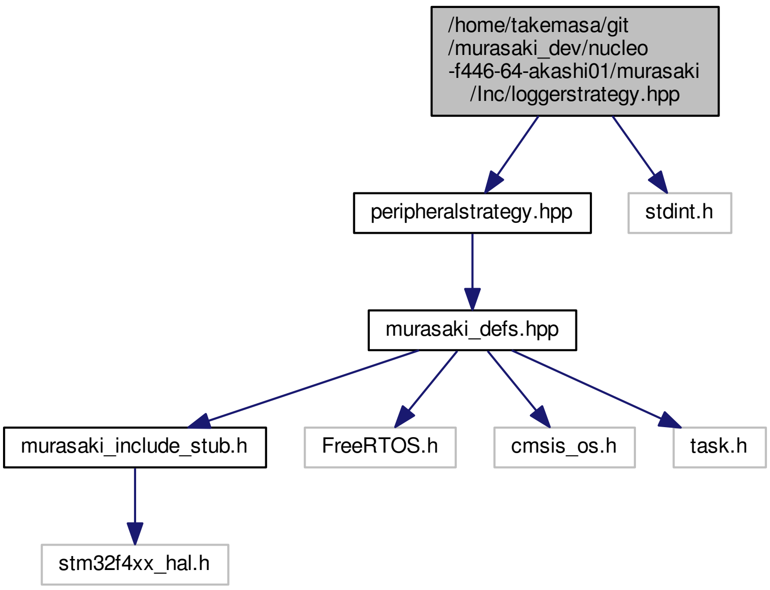
Author

Seiichi "Suikan" Horie

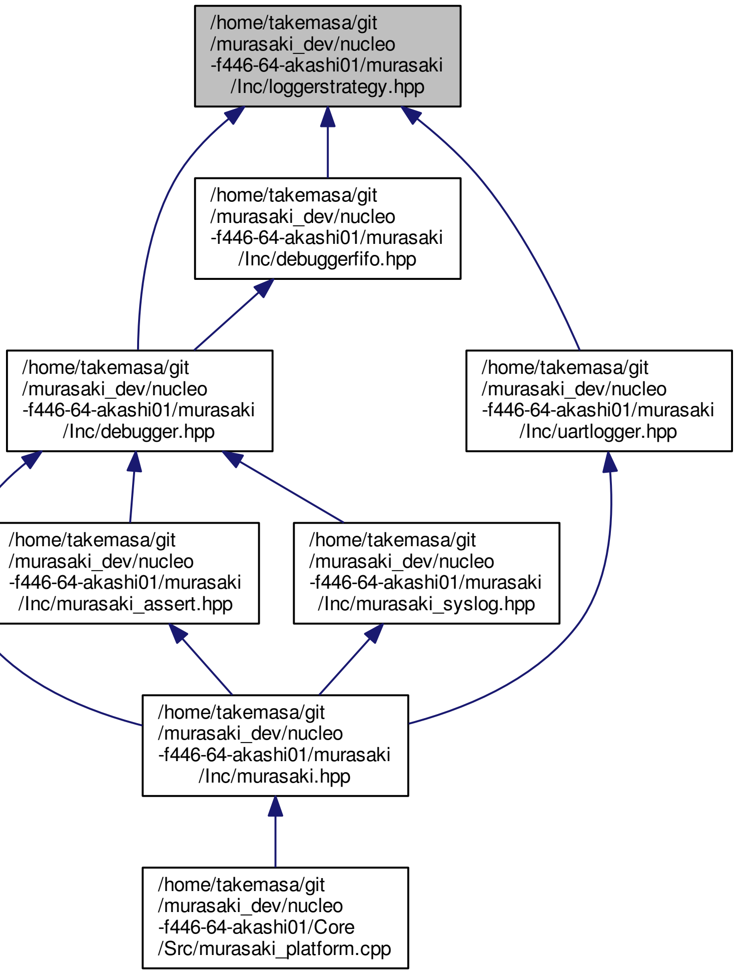
13.27 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/loggerstrategy.hpp File Reference

```
#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

Namespaces

- murasaki

13.27.1 Detailed Description

Date

2018/01/20

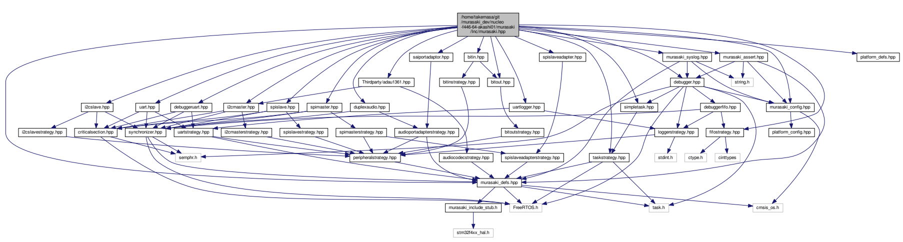
Author

: Seiichi "Suikan" Horie

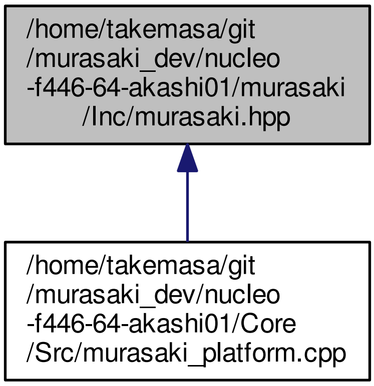
13.28 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki.hpp File Reference

```
#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 "saiportadaptor.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:



13.28.1 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.

13.29 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_0_intro.hpp File Reference

13.29.1 Detailed Description

Date

2018/02/01

Author

Seiichi "Suikan" Horie

13.30 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_1_env.hpp File Reference

13.30.1 Detailed Description

Date

2018/02/01

Author

Seiichi "Suikan" Horie

13.31 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_2_ug.hpp File Reference

13.31.1 Detailed Description

Date

2018/02/01

Author

Seiichi "Suikan" Horie

13.32 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_3_pg.hpp File Reference

13.32.1 Detailed Description

Date

May 25, 2018

Author

Seiichi "Suikan" Horie

13.33 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_4_mod.hpp File Reference

13.33.1 Detailed Description

Date

May 25, 2018

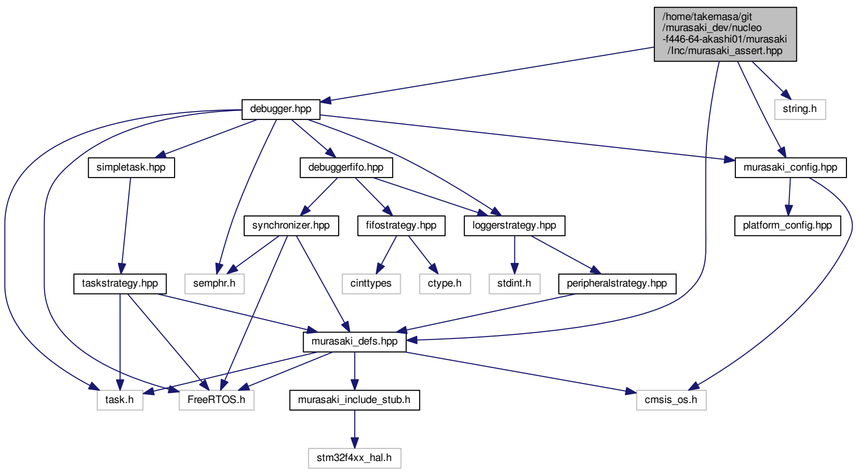
Author

Seiichi "Suikan" Horie

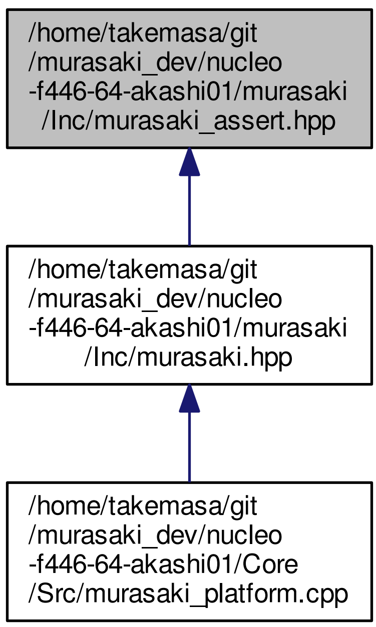
13.34 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_assert.hpp File Reference

```
#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

Macros

- #define MURASAKI_ASSERT(COND)
- #define MURASAKI_PRINT_ERROR(ERR)

13.34.1 Detailed Description

Date

2018/01/31

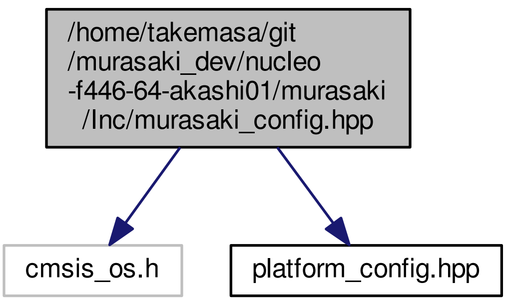
Author

Seiichi "Suikan" Horie

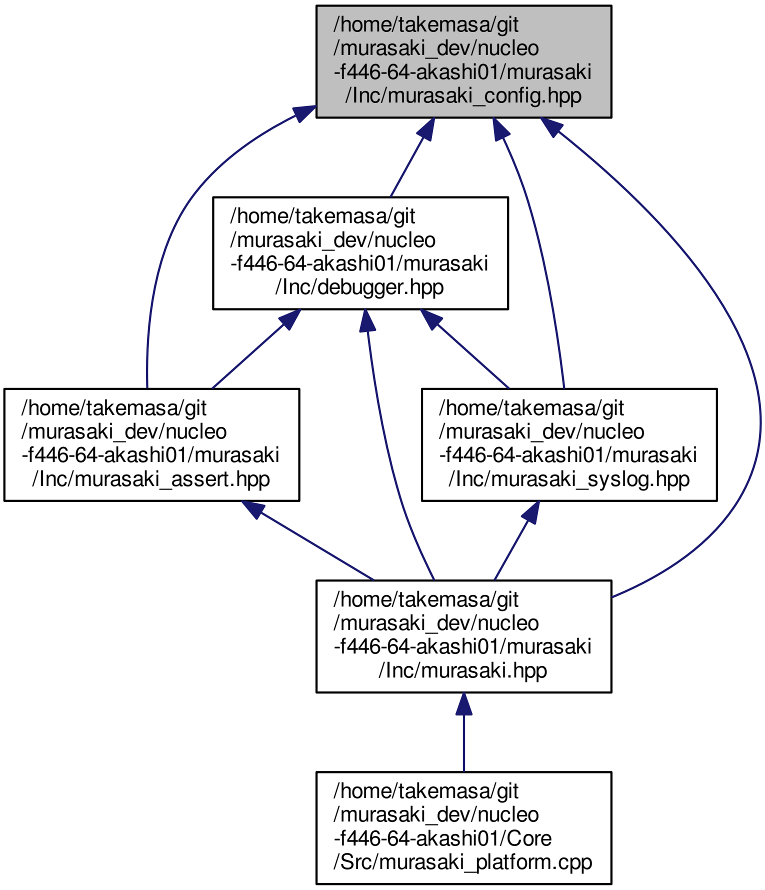
13.35 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_config.hpp File Reference

```
#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
- #define PLATFORM_CONFIG_DEBUG_BUFFER_SIZE 4096
- #define PLATFORM_CONFIG_DEBUG_SERIAL_TIMEOUT (murasaki::kwmsIndefinitely)
- #define PLATFORM_CONFIG_DEBUG_TASK_STACK_SIZE 256
- #define PLATFORM_CONFIG_DEBUG_TASK_PRIORITY murasaki::ktpHigh
- #define MURASAKI_CONFIG_NODEBUG false
- #define MURASAKI_CONFIG_NOCYCCNT false

13.35.1 Detailed Description

Date

2018/01/03

Author

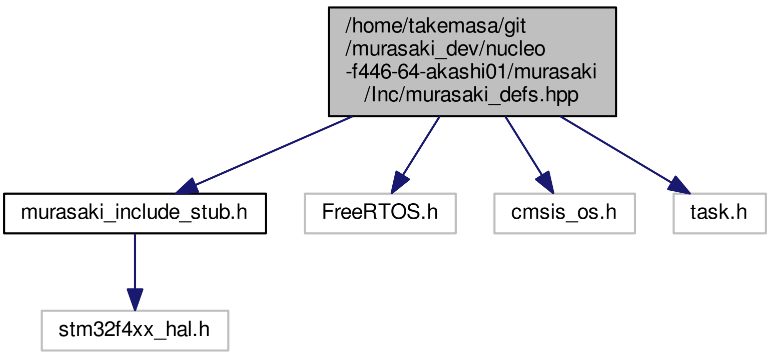
Seiichi "Suikan" Horie

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

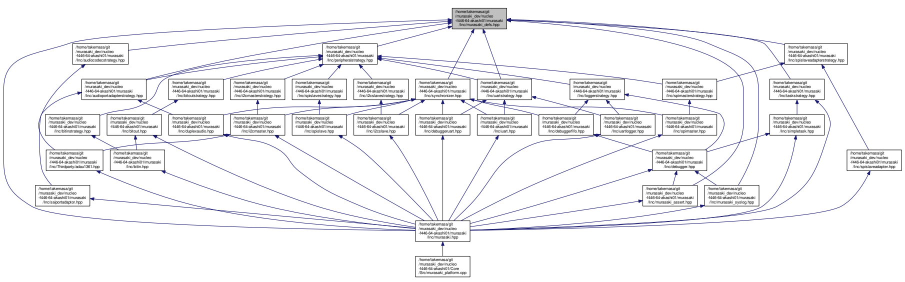
13.36 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_defs.hpp File Reference

```
#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

Enumerations

Functions

- static bool murasaki::IsTaskContext ()
- static void murasaki::CleanAndInvalidateDataCacheByAddress (void ∗address, size_t size)
- static void murasaki::CleanDataCacheByAddress (void ∗address, size_t size)
- void murasaki::InitCycleCounter ()
- unsigned int murasaki::GetCycleCounter ()
- static void murasaki::Sleep (unsigned int duration_ms)

13.36.1 Detailed Description

Date

2017/11/05

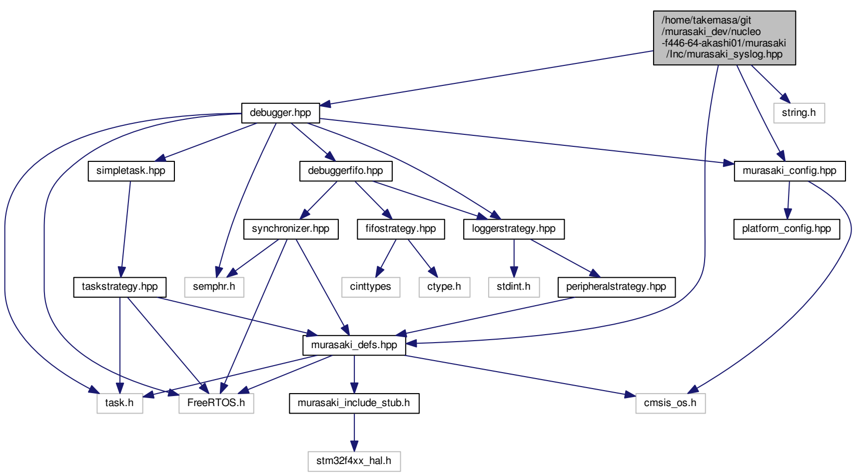
Author

Seiichi "Suikan" Horie

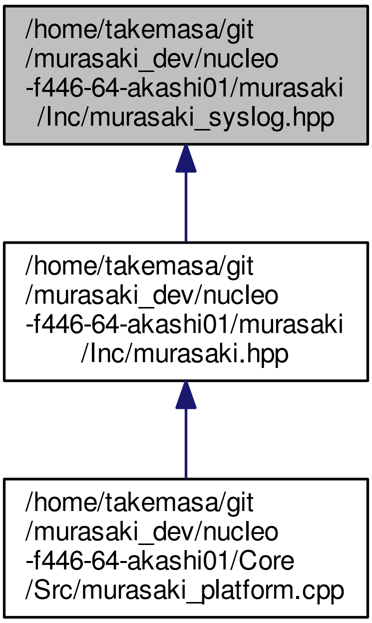
13.37 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/murasaki_syslog.hpp File Reference

```
#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

Macros

- #define MURASAKI_SYSLOG(OBJPTR, FACILITY, SEVERITY, FORMAT, ...)

Functions

- void murasaki::SetSyslogSererityThreshold (murasaki::SyslogSeverity severity)
- void murasaki::SetSyslogFacilityMask (uint32_t mask)
- void murasaki::AddSyslogFacilityToMask (murasaki::SyslogFacility facility)
- void murasaki::RemoveSyslogFacilityFromMask (murasaki::SyslogFacility facility)
- bool murasaki::AllowedSyslogOut (murasaki::SyslogFacility facility, murasaki::SyslogSeverity severity)

13.37.1 Detailed Description

Date

2018/09/01

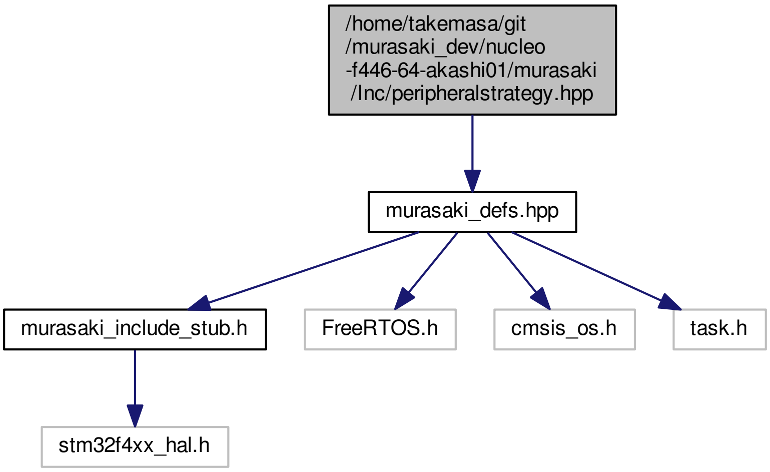
Author

Seiichi "Suikan" Horie

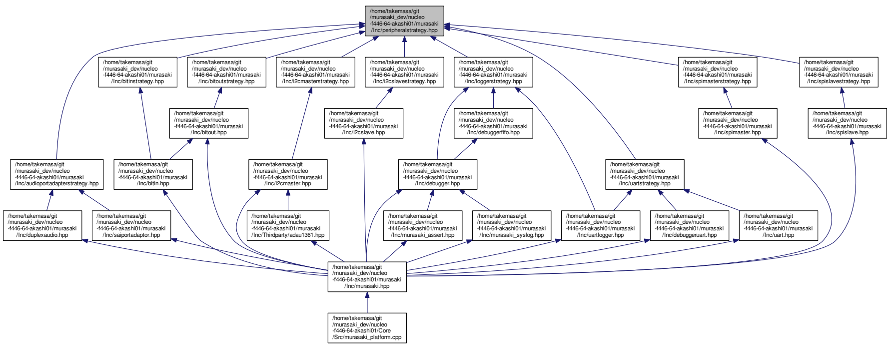
13.38 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/peripheralstrategy.hpp File Reference

`#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

Namespaces

- murasaki

13.38.1 Detailed Description

Date

2018/04/26

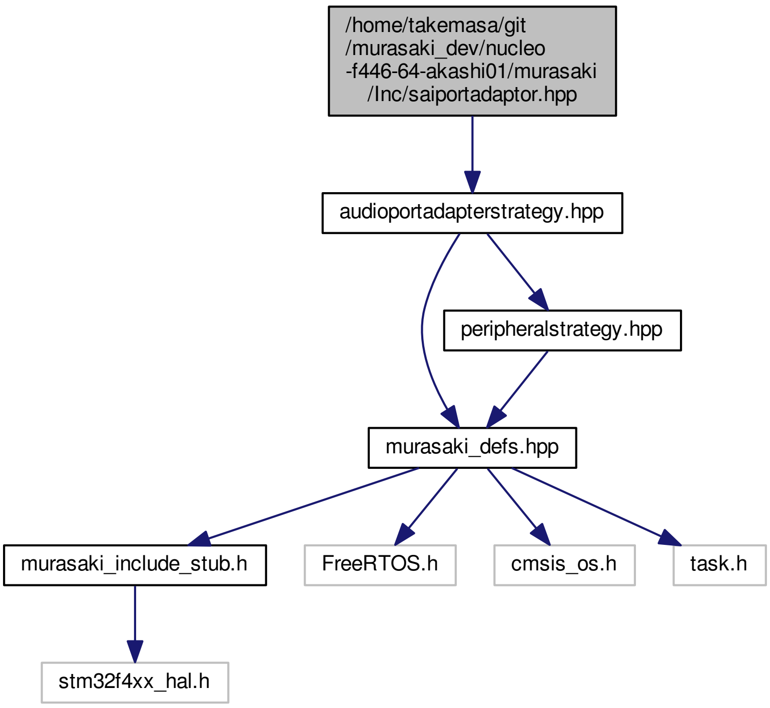
Author

: Seiichi "Suikan" Horie

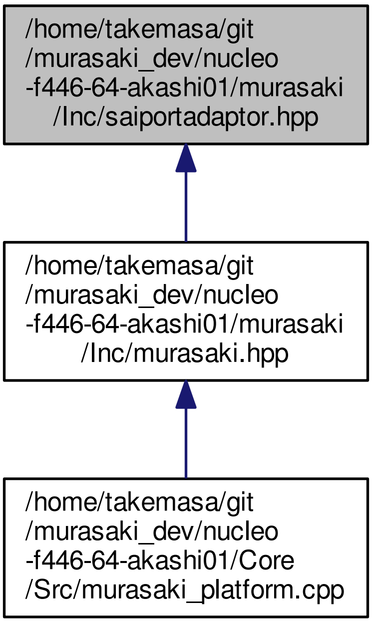
13.39 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/saiportadaptor.hpp File Reference

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

Include dependency graph for saiportadaptor.hpp:



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



Classes

- class murasaki::SaiPortAdaptor

Namespaces

- murasaki

13.39.1 Detailed Description

Date

2019/07/28

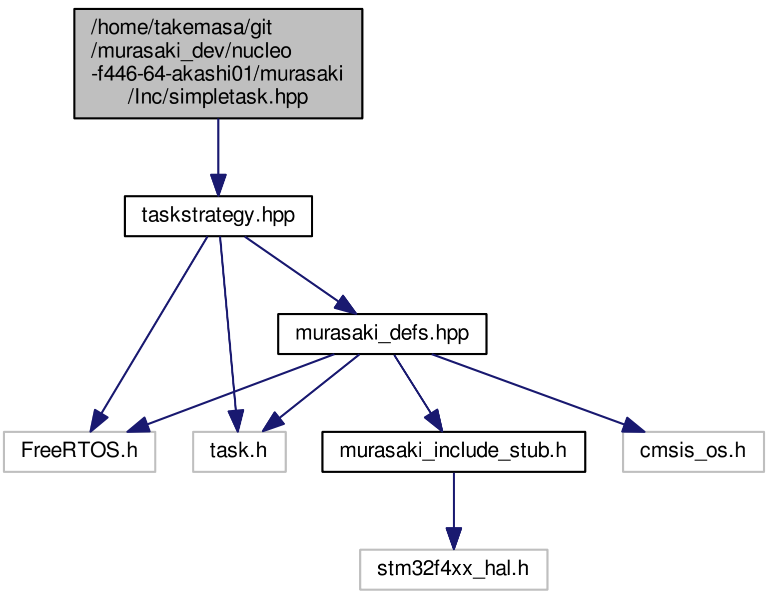
Author

takemasa

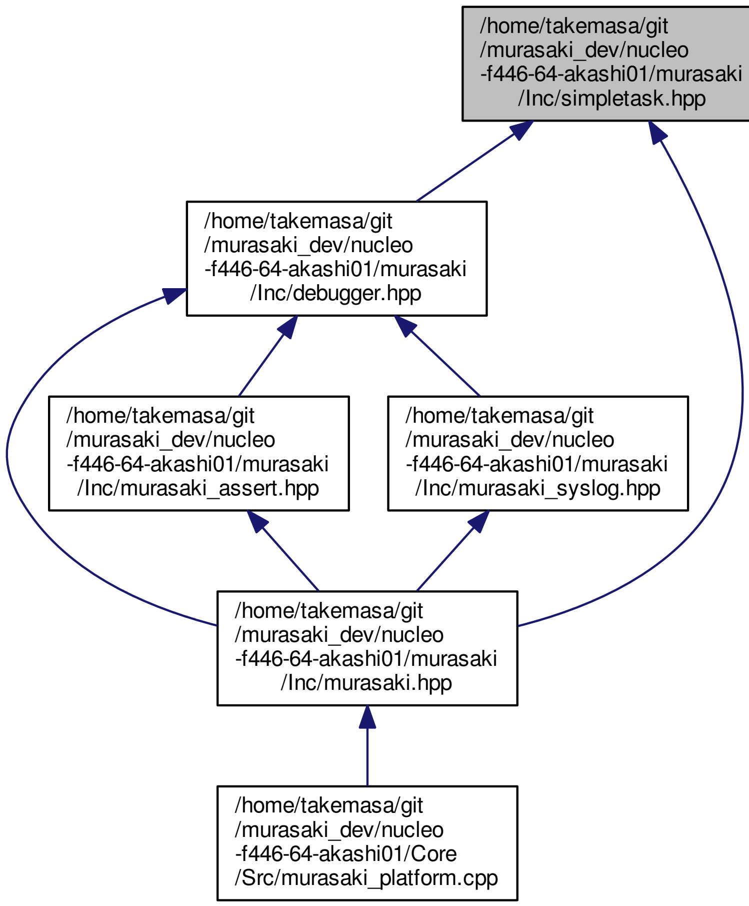
13.40 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/simpletask.hpp File Reference

`#include <taskstrategy.hpp>`

Include dependency graph for simpletask.hpp:



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



Classes

- class murasaki::SimpleTask

Namespaces

- murasaki

13.40.1 Detailed Description

Date

2019/02/03

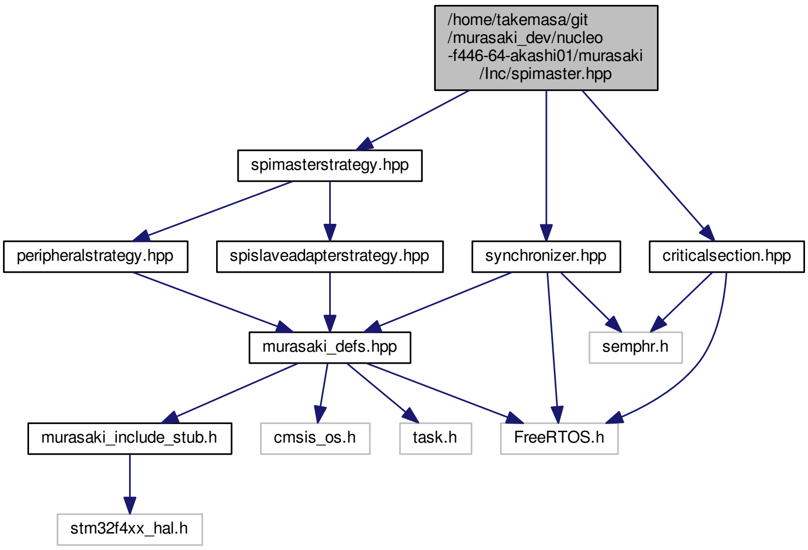
Author

Seiichi "Suikan" Horie

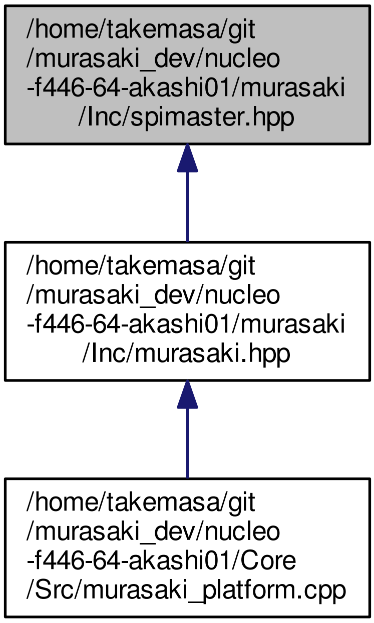
13.41 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spimaster.hpp File Reference

```
#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

Namespaces

- murasaki

13.41.1 Detailed Description

Date

2018/02/14

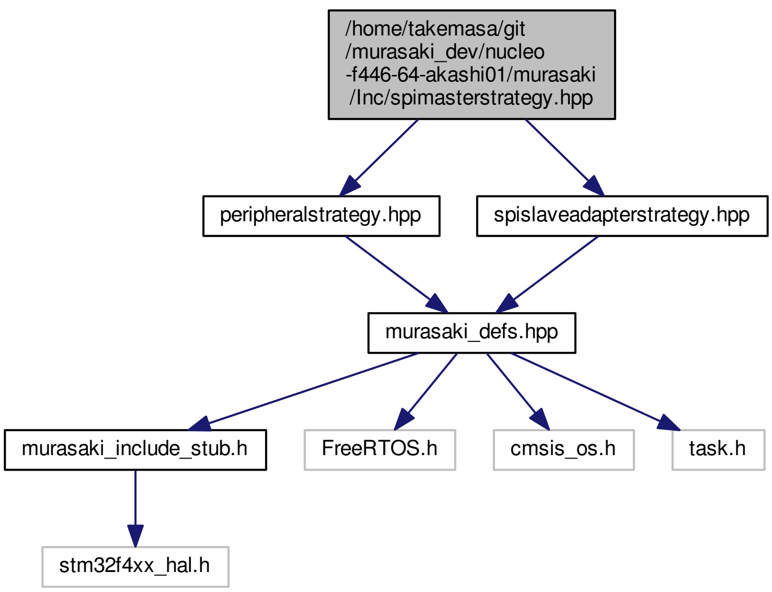
Author

Seiichi "Suikan" Horie

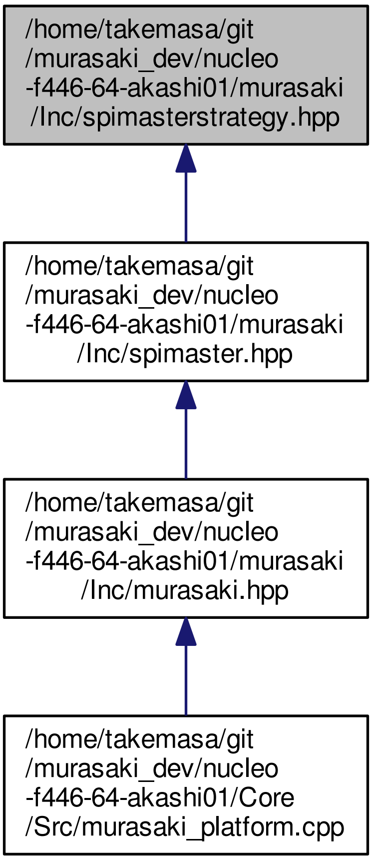
13.42 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spimasterstrategy.hpp File Reference

```
#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

Namespaces

- murasaki

13.42.1 Detailed Description

Date

2018/02/11

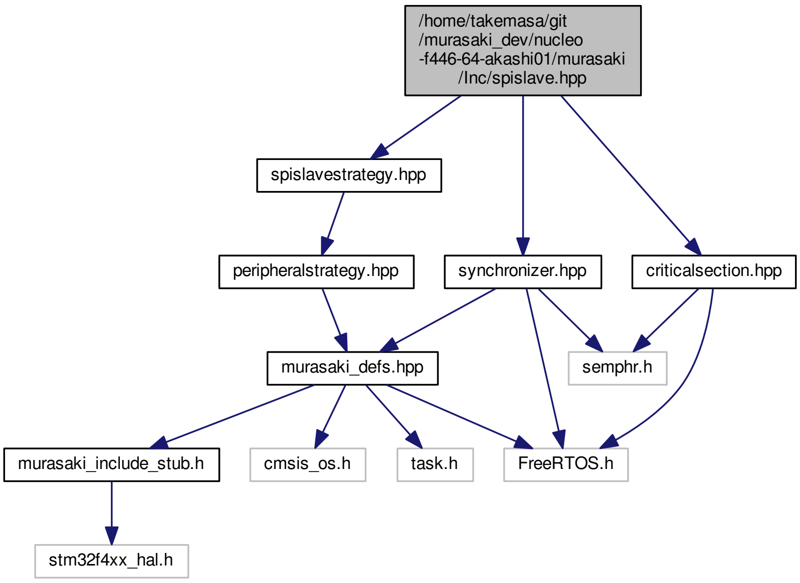
Author

: Seiichi "Suikan" Horie

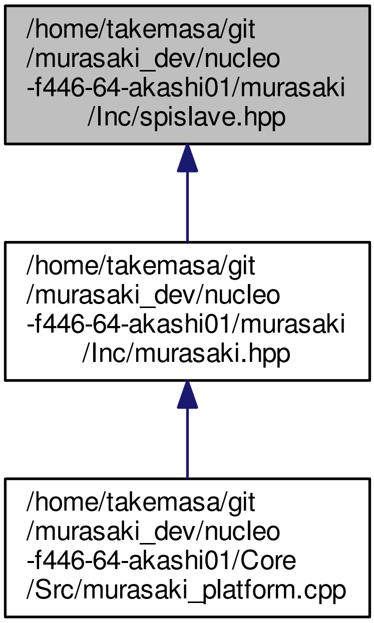
13.43 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spislave.hpp File Reference

```
#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

Namespaces

- murasaki

13.43.1 Detailed Description

Date

2018/02/14

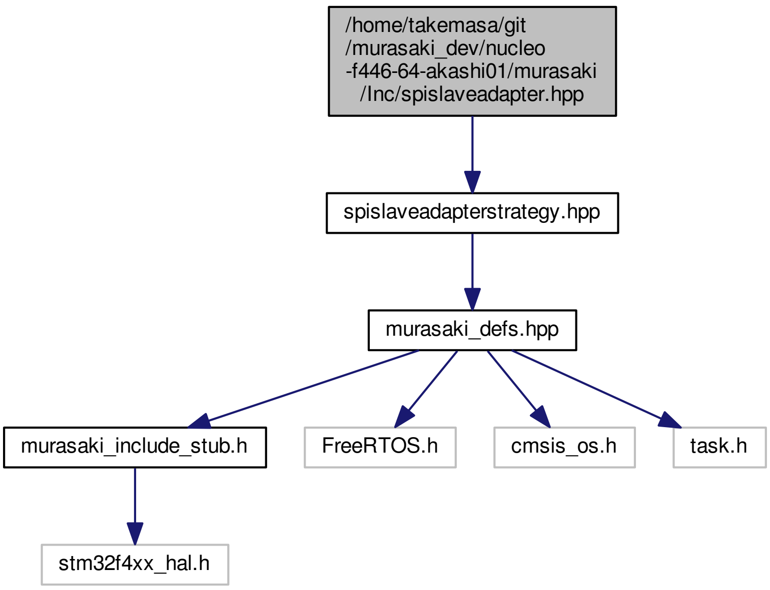
Author

Seiichi "Suikan" Horie

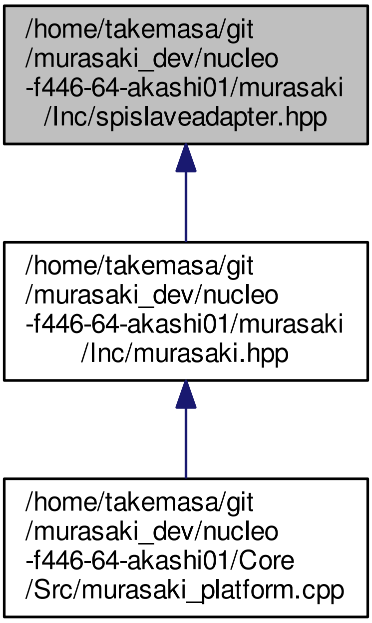
13.44 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spislaveadapter.hpp File Reference

```
#include <spislaveadapterstrategy.hpp>
```

Include dependency graph for spislaveadapter.hpp:



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



Classes

- class murasaki::SpiSlaveAdapter

Namespaces

- murasaki

13.44.1 Detailed Description

Date

2018/02/17

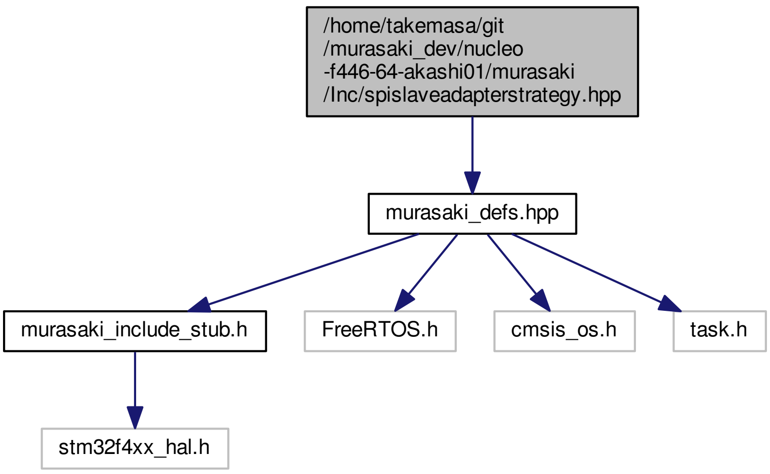
Author

Seiichi "Suikan" Horie

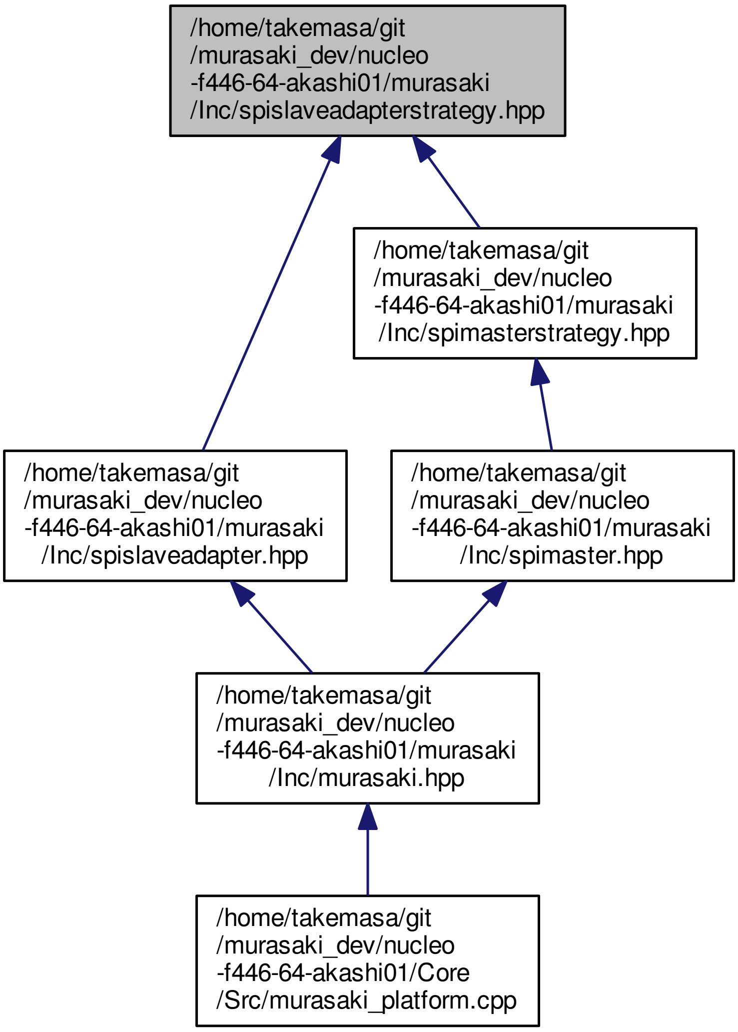
13.45 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spislaveadapterstrategy File Reference

```
#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

Namespaces

- murasaki

13.45.1 Detailed Description

Date

2018/02/11

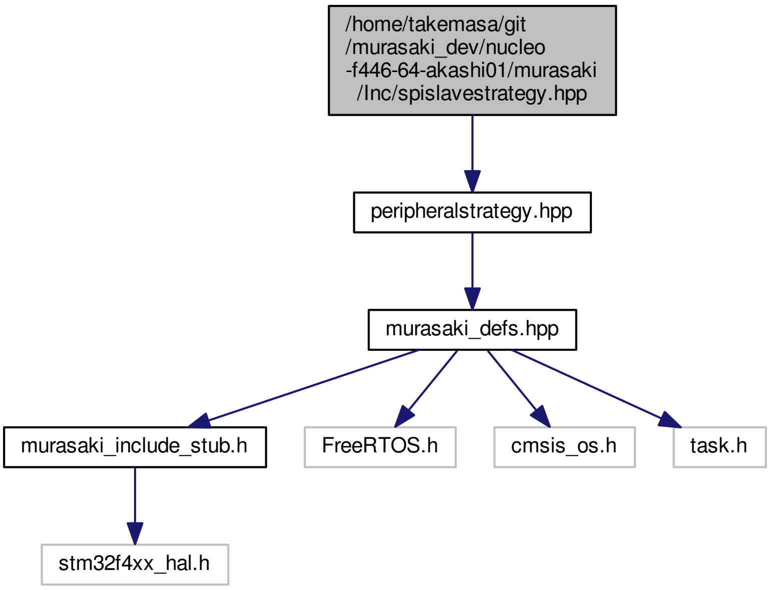
Author

: Seiichi "Suikan" Horie

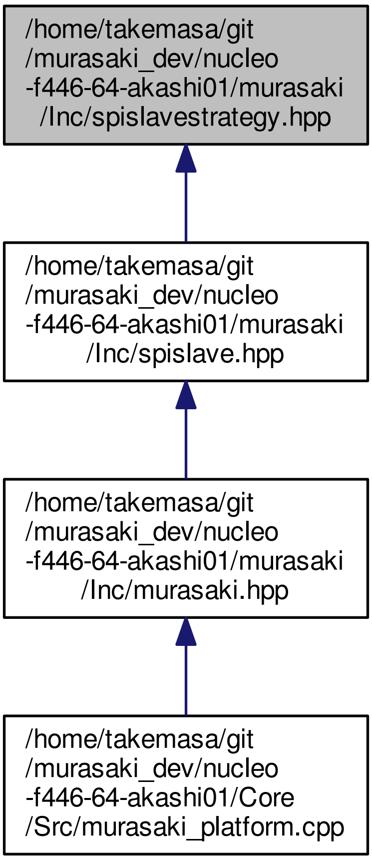
13.46 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/spislavestrategy.hpp File Reference

```
#include <peripheralstrategy.hpp>
```

Include dependency graph for spislavestrategy.hpp:



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



Classes

- class murasaki::SpiSlaveStrategy

Namespaces

- murasaki

13.46.1 Detailed Description

Date

2018/02/11

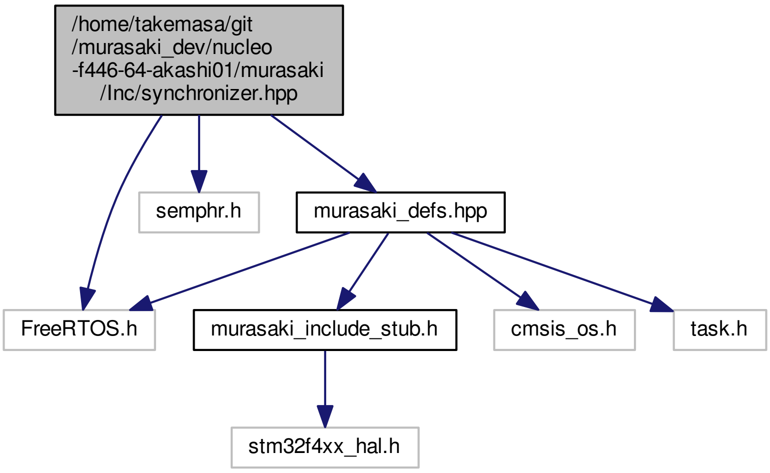
Author

: Seiichi "Suikan" Horie

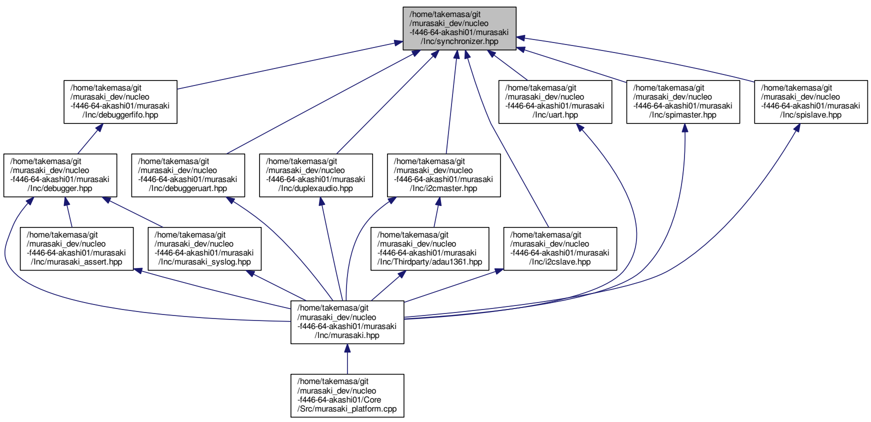
13.47 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/synchronizer.hpp File Reference

```
#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

Namespaces

- murasaki

13.47.1 Detailed Description

Date

2018/01/26

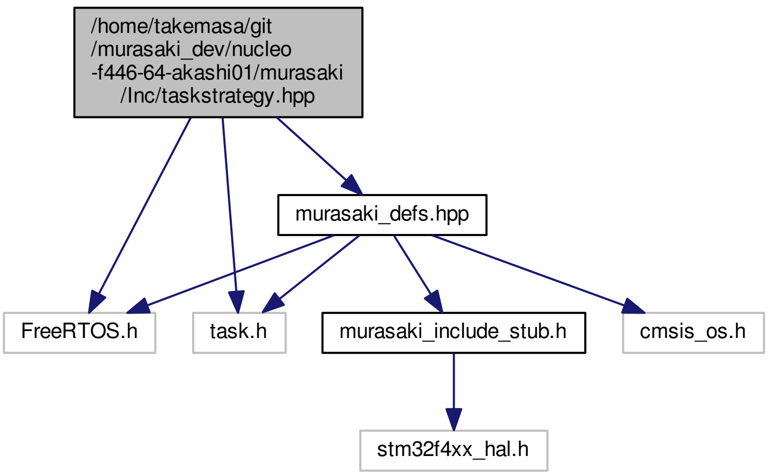
Author

Seiichi "Suikan" Horie

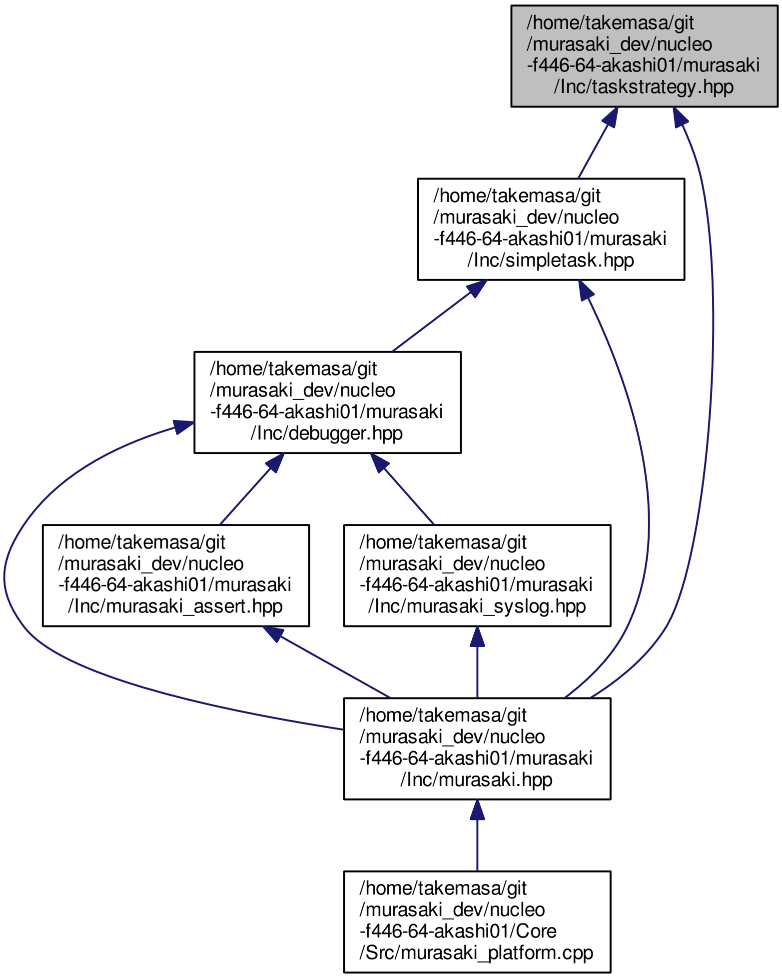
13.48 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/taskstrategy.hpp File Reference

```
#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

Namespaces

- murasaki

13.48.1 Detailed Description

Date

2018/02/20

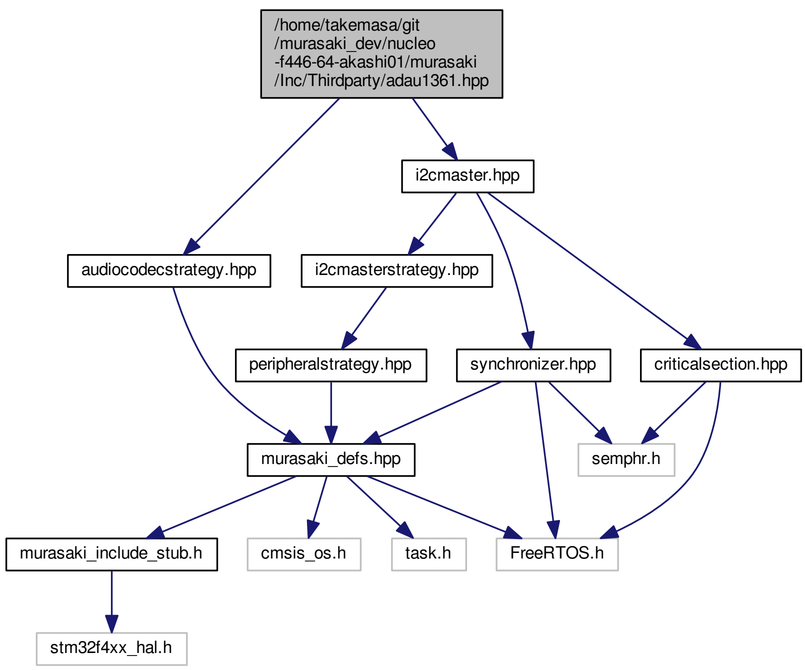
Author

: Seiichi "Suikan" Horie

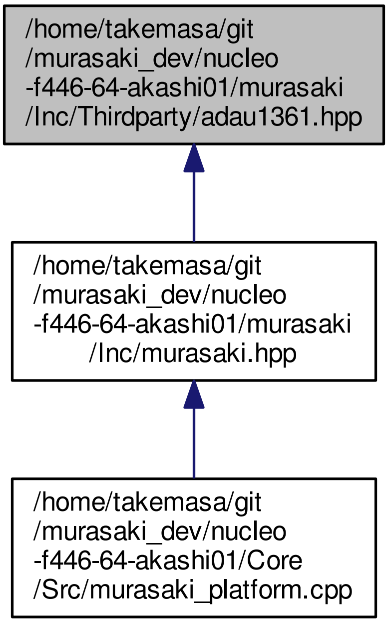
13.49 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/Thirdparty/adau1361.hpp File Reference

```
#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

Namespaces

- murasaki

13.49.1 Detailed Description

Date

2018/05/11

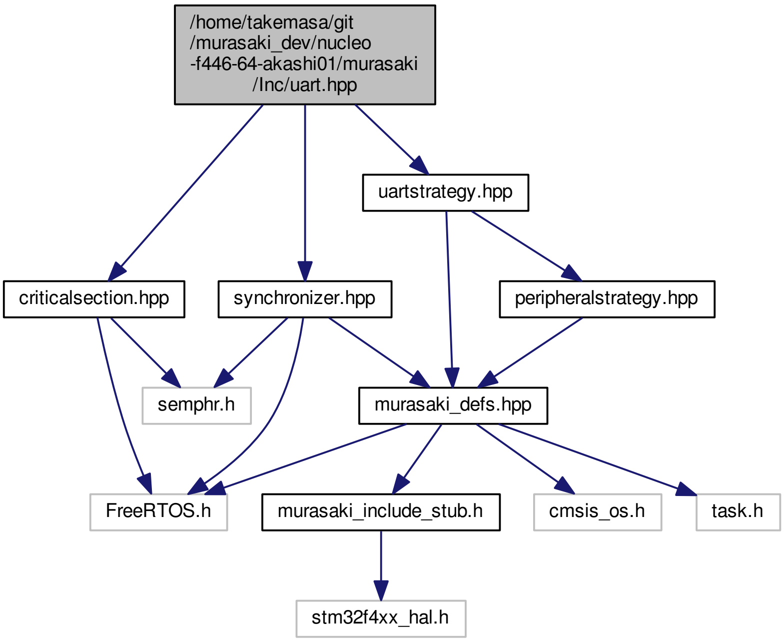
Author

: Seiichi "Suikan" Horie

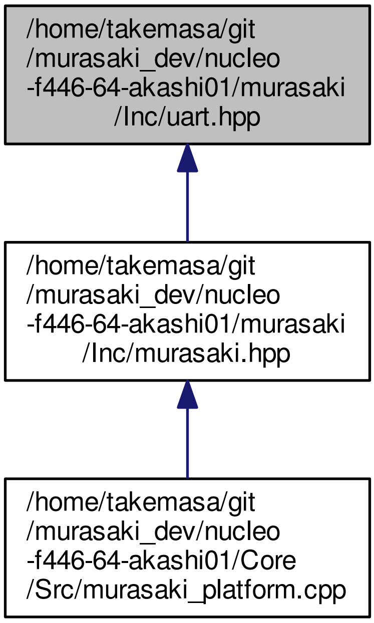
13.50 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/uart.hpp File Reference

```
#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

Namespaces

- murasaki

13.50.1 Detailed Description

Date

2017/11/05

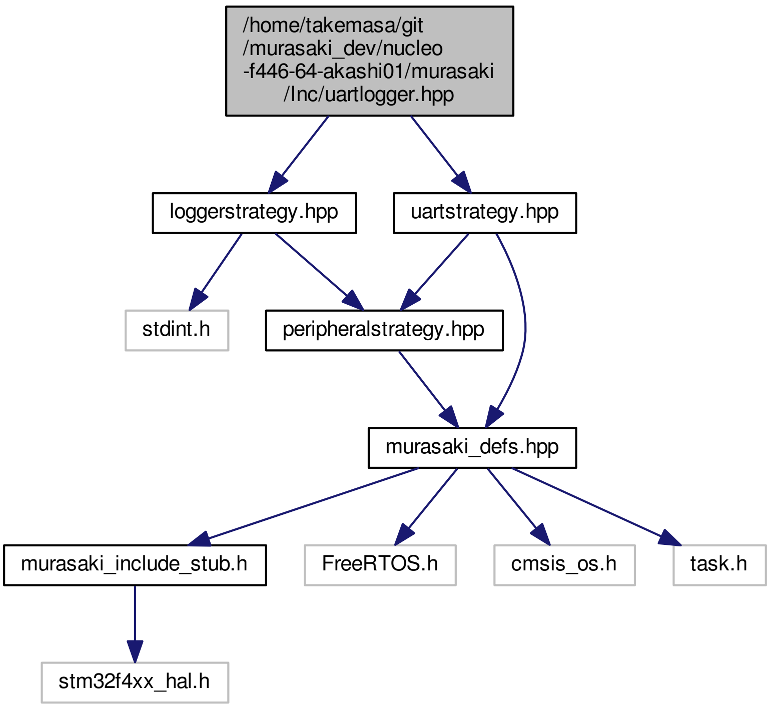
Author

Seiichi "Suikan" Horie

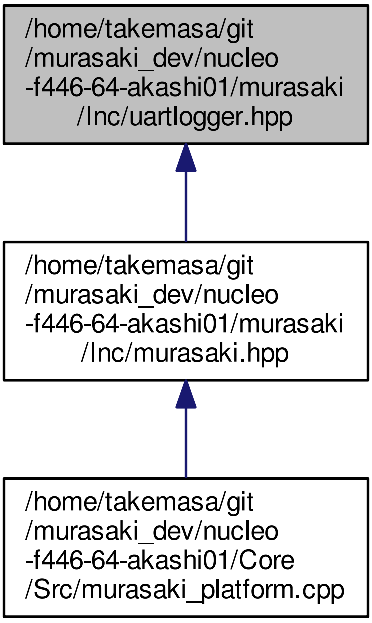
13.51 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/uartlogger.hpp File Reference

```
#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

Namespaces

- murasaki

13.51.1 Detailed Description

Date

2018/01/20

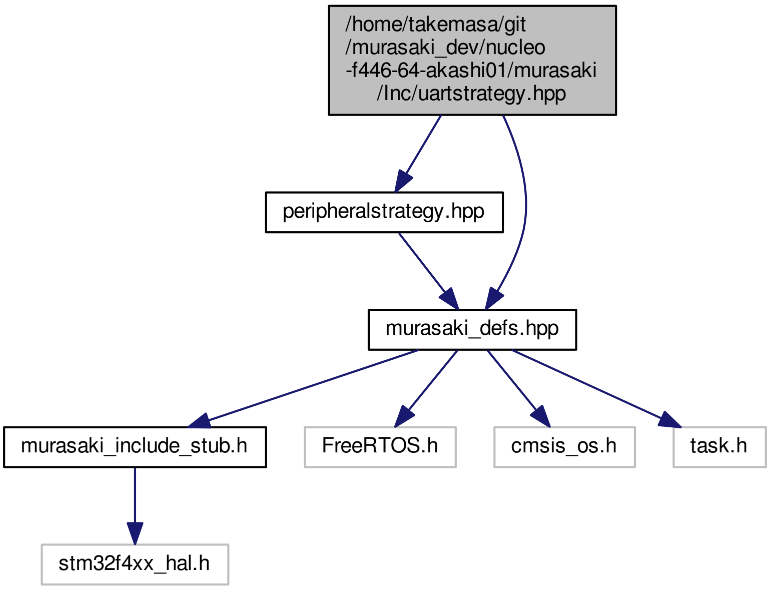
Author

: Seiichi "Suikan" Horie

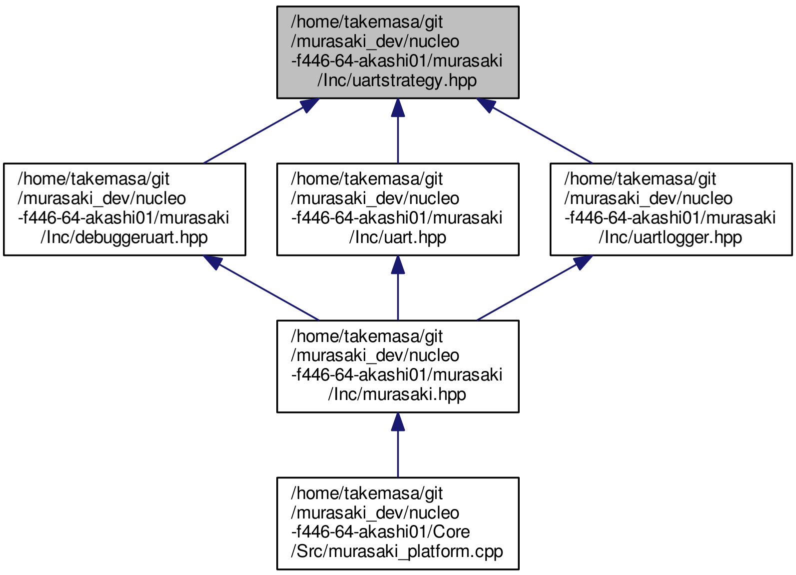
13.52 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Inc/uartstrategy.hpp File Reference

```
#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

Namespaces

- murasaki

13.52.1 Detailed Description

Date

2017/11/04

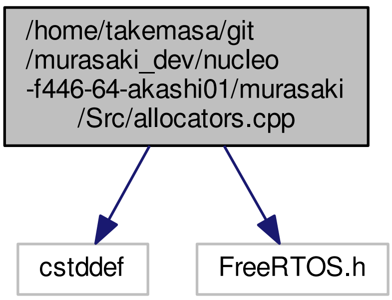
Author

: Seiichi "Suikan" Horie

13.53 /home/takemasa/git/murasaki_dev/nucleo-f446-64-akashi01/murasaki/Src/allocators.cpp File Reference

```
#include <cstddef>
#include <FreeRTOS.h>
```

Include dependency graph for allocators.cpp:



Functions

- void ∗ operator new (std::size_t size)
- void ∗ operator new[ ] (std::size_t size)
- void operator delete (void ∗ptr)
- void operator delete[ ] (void ∗ptr)

13.53.1 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().

Index