

mthread

0.5

Generated by Doxygen 1.7.6.1

Tue Jul 3 2012 14:07:17

Contents

1	Class Index	1
1.1	Class Hierarchy	1
2	Class Index	3
2.1	Class List	3
3	File Index	5
3.1	File List	5
4	Class Documentation	7
4.1	EventHandler Class Reference	7
4.1.1	Detailed Description	8
4.1.2	Member Function Documentation	8
4.1.2.1	condition	8
4.1.2.2	loop	8
4.1.2.3	on_event	8
4.2	SwitchInput Class Reference	9
4.2.1	Detailed Description	10
4.2.2	Member Enumeration Documentation	10
4.2.2.1	Type	10
4.2.3	Constructor & Destructor Documentation	10
4.2.3.1	SwitchInput	10
4.2.4	Member Function Documentation	11
4.2.4.1	is_closed	11
4.2.4.2	is_open	11
4.2.4.3	loop	11

4.2.4.4	time_closed	11
4.2.4.5	time_open	11
4.3	Thread Class Reference	12
4.3.1	Detailed Description	13
4.3.2	Member Enumeration Documentation	13
4.3.2.1	Mode	13
4.3.3	Member Function Documentation	14
4.3.3.1	call	14
4.3.3.2	get_mode	14
4.3.3.3	kill	14
4.3.3.4	loop	15
4.3.3.5	pause	15
4.3.3.6	resume	15
4.3.3.7	sleep	16
4.3.3.8	sleep_micro	16
4.3.3.9	sleep_milli	16
4.4	ThreadList Class Reference	17
4.4.1	Detailed Description	18
4.4.2	Constructor & Destructor Documentation	18
4.4.2.1	ThreadList	18
4.4.3	Member Function Documentation	18
4.4.3.1	add_thread	18
4.4.3.2	loop	18
5	File Documentation	21
5.1	mthread.cpp File Reference	21
5.1.1	Detailed Description	21
5.2	mthread.h File Reference	21
5.2.1	Detailed Description	22

Chapter 1

Class Index

1.1 Class Hierarchy

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

Thread	12
EventHandler	7
SwitchInput	9
ThreadList	17

Chapter 2

Class Index

2.1 Class List

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

EventHandler	A Thread that only executes when a given event occurs	7
SwitchInput	Handler for a switch input	9
Thread	Provides thread functionality	12
ThreadList	An object for running several Thread objects simultaneously. A - ThreadList object is a Thread in and of itself. This allows the creation of tiered ThreadList objects by placing a lower-priority ThreadList inside of a higher-priority ThreadList	17

Chapter 3

File Index

3.1 File List

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

mthread.cpp	21
mthread.h	21

Chapter 4

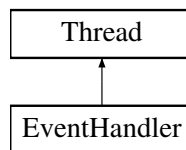
Class Documentation

4.1 EventHandler Class Reference

A [Thread](#) that only executes when a given event occurs.

```
#include <mthread.h>
```

Inheritance diagram for EventHandler:



Public Member Functions

- [EventHandler](#) ()
Constructor.
- virtual [~EventHandler](#) ()
Destructor.

Protected Member Functions

- virtual bool [condition](#) ()
*Condition to be evaluated on each call to determine whether or not to run the event.
The condition will not be checked again until the [on_event\(\)](#) function returns false.*
- bool [loop](#) ()
Main loop.
- virtual bool [on_event](#) ()
Called when the even occurs.

Private Attributes

- bool [trigger](#)

Becomes true when the event occurs; returns to false when the event has been handled.

4.1.1 Detailed Description

A [Thread](#) that only executes when a given event occurs.

4.1.2 Member Function Documentation

4.1.2.1 `bool EventHandler::condition ()` [protected, virtual]

Condition to be evaluated on each call to determine whether or not to run the event. The condition will not be checked again until the [on_event\(\)](#) function returns false.

Returns

true if the event has occurred; false if it has not.

4.1.2.2 `bool EventHandler::loop (void)` [protected, virtual]

Main loop.

See also

[Thread::loop\(\)](#).

Reimplemented from [Thread](#).

4.1.2.3 `bool EventHandler::on_event ()` [protected, virtual]

Called when the even occurs.

Returns

true if the event hasn't been completely handled, false to wait for the next event.

Note

This function does not necessarily have to check the `kill_flag` value as it will be honoured by the [loop\(\)](#) function when the `trigger` value is false.

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

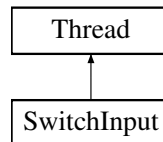
- [mthread.h](#)
- [mthread.cpp](#)

4.2 SwitchInput Class Reference

Handler for a switch input.

```
#include <mthread.h>
```

Inheritance diagram for SwitchInput:



Public Types

- enum `Type` { `pull_up_internal`, `pull_up`, `pull_down` }
Types of switches.

Public Member Functions

- `SwitchInput` (int `pin`, unsigned long `debounce`=`DEFAULT_DEBOUNCE`, `Type` `type`=`pull_up_internal`)
Constructor.
- virtual `~SwitchInput` ()
Destructor.
- bool `is_closed` () const
Checks to see if the switch is closed.
- bool `is_open` () const
Checks to see if the switch is open.
- virtual void `on_close` ()
Called once when the switch closes.
- virtual void `on_open` ()
Called once when the switch opens.
- unsigned long `time_closed` () const
Checks the length of time the switch has been closed.
- unsigned long `time_open` () const
Checks the length of time the switch has been open.

Protected Member Functions

- bool `loop` ()
Main loop.

Private Attributes

- unsigned long [debounce](#)
The debounce time (in milliseconds).
- unsigned long [last_change](#)
The time of the last change (in milliseconds, ignoring debounce).
- unsigned long [last_debounce](#)
The time of the last change (in milliseconds, after debounce filtering).
- int [current_value](#)
The switch's current value (after debounce filtering).
- int [last_value](#)
The switch's value on the last read (ignoring debounce).
- int [pin](#)
The pin the switch is connected to.
- [Type](#) [type](#)
The type of switch connected.

4.2.1 Detailed Description

Handler for a switch input.

4.2.2 Member Enumeration Documentation

4.2.2.1 enum `SwitchInput::Type`

Types of switches.

Enumerator:

- `pull_up_internal`** Switch uses the internal pull-up resistor.
- `pull_up`** Switch uses an external pull-up resistor.
- `pull_down`** Switch uses an external pull-down resistor.

4.2.3 Constructor & Destructor Documentation

4.2.3.1 `SwitchInput::SwitchInput (int pin, unsigned long debounce = DEFAULT_DEBOUNCE, Type type = pull_up_internal)`

Constructor.

Parameters

<i>pin</i>	The pin number the switch is connected to.
<i>debounce</i>	The debounce time (in milliseconds).
<i>type</i>	The type of switch connected.

4.2.4 Member Function Documentation

4.2.4.1 `bool SwitchInput::is_closed () const`

Checks to see if the switch is closed.

Returns

true if the switch is closed, false if it's open.

4.2.4.2 `bool SwitchInput::is_open () const`

Checks to see if the switch is open.

Returns

true if the switch is open, false if it's closed.

4.2.4.3 `bool SwitchInput::loop (void) [protected, virtual]`

Main loop.

See also

[Thread::loop\(\)](#).

Reimplemented from [Thread](#).

4.2.4.4 `unsigned long SwitchInput::time_closed () const`

Checks the length of time the switch has been closed.

Returns

The amount of time (in milliseconds) the switch has been closed, or 0 if the switch is open.

4.2.4.5 `unsigned long SwitchInput::time_open () const`

Checks the length of time the switch has been open.

Returns

The amount of time (in milliseconds) the switch has been open, or 0 if the switch is closed.

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

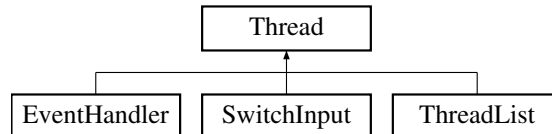
- [mthread.h](#)
- [mthread.cpp](#)

4.3 Thread Class Reference

Provides thread functionality.

```
#include <mthread.h>
```

Inheritance diagram for Thread:



Public Types

- enum [Mode](#) { [run_mode](#), [pause_mode](#), [sleep_mode](#), [sleep_milli_mode](#), [sleep_micro_mode](#), [kill_mode](#) }

Various running modes for a [Thread](#).

Public Member Functions

- [Thread](#) ()
Constructor.
- virtual [~Thread](#) ()
Destructor.
- [Mode](#) [get_mode](#) () const
Returns the running mode for the [Thread](#).
- bool [kill](#) (bool force=false)
Kills a [Thread](#).
- bool [pause](#) ()
Pauses a [Thread](#). This function will cause a [Thread](#) to pause until its [resume\(\)](#) function is called. This function will cancel any sleep timer currently in effect.
- bool [resume](#) ()
Resumes a paused or sleeping [Thread](#).
- bool [sleep](#) (unsigned long t)
Puts the [Thread](#) to sleep for a certain number of seconds. If already running, the thread's [loop\(\)](#) function will be allowed to finish, but will not be called again until the timeout has expired, or the [resume\(\)](#) or [kill\(\)](#) function has been called.
- bool [sleep_micro](#) (unsigned long t)
Puts the [Thread](#) to sleep for a certain number of microseconds. If already running, the thread's [loop\(\)](#) function will be allowed to finish, but will not be called again until the timeout has expired, or the [resume\(\)](#) or [kill\(\)](#) function has been called.
- bool [sleep_milli](#) (unsigned long t)
Puts the [Thread](#) to sleep for a certain number of milliseconds. If already running, the thread's [loop\(\)](#) function will be allowed to finish, but will not be called again until the timeout has expired, or the [resume\(\)](#) or [kill\(\)](#) function has been called.

Protected Member Functions

- virtual bool `loop()`

The `Thread`'s main loop. This function is to be overridden. It takes the place of the `loop` function found in most Arduino programs. A single `loop()` should run as quickly as possible, as it will hold up other `Thread` objects while it is executing.

Protected Attributes

- bool `kill_flag`

Kill flag. This variable should be checked at the beginning of every `loop()` function. If it is true, the `Thread` has been requested to be killed, and the `loop()` function should behave accordingly. The request can be denied by resetting it to false.

Private Member Functions

- bool `call()`

Determines if the function is active and runs through a loop if appropriate. This function is called automatically by a `ThreadList`.

Private Attributes

- unsigned long `stop_time`

The time the thread was stopped at.

- unsigned long `wait_time`

The amount of the the thread is to wait for.

- `Mode mode`

The thread's running mode (can be read through the `get_mode()` function).

Friends

- class `ThreadList`
- void `loop()` (void)

4.3.1 Detailed Description

Provides thread functionality.

4.3.2 Member Enumeration Documentation

4.3.2.1 enum `Thread::Mode`

Various running modes for a `Thread`.

Enumerator:

run_mode [Thread](#) is running.
pause_mode [Thread](#) is paused.
sleep_mode [Thread](#) is sleeping (for seconds).
sleep_milli_mode [Thread](#) is sleeping (for milliseconds).
sleep_micro_mode [Thread](#) is sleeping (for microseconds).
kill_mode [Thread](#) is to be killed on next call.

4.3.3 Member Function Documentation

4.3.3.1 `bool Thread::call() [private]`

Determines if the function is active and runs through a loop if appropriate. This function is called automatically by a [ThreadList](#).

Returns

true if the [Thread](#) needs to be called again, false if the [Thread](#) has completed execution.

Note

It is important to note that once a [Thread](#) has completed its execution it will automatically destroy itself and MUST NOT be used again. A new instance must first be created.

4.3.3.2 `Thread::Mode Thread::get_mode() const`

Returns the running mode for the [Thread](#).

Returns

The running mode.

4.3.3.3 `bool Thread::kill(bool force = false)`

Kills a [Thread](#).

Parameters

<i>force</i>	If true, the Thread will be killed immediately on the next call without running any more loops, if false, the Thread will have to opportunity to terminate cleanly but will be resumed if sleeping or paused.
--------------	---

Note

If the force parameter is set to false, the [Thread](#) could possibly ignore or cancel the request, however this is still the preferred way of calling the [kill\(\)](#) function.

Returns

true on success, false on failure.

4.3.3.4 bool Thread::loop(void) [protected, virtual]

The [Thread](#)'s main loop. This function is to be overridden. It takes the place of the loop function found in most Arduino programs. A single [loop\(\)](#) should run as quickly as possible, as it will hold up other [Thread](#) objects while it is executing.

Note

At the beginning of each loop, the function should check the `kill_flag`.

Returns

true if the loop needs to be called again, false if the [Thread](#) has completed executing (at which point it will be destroyed).

Reimplemented in [SwitchInput](#), [EventHandler](#), and [ThreadList](#).

4.3.3.5 bool Thread::pause ()

Pauses a [Thread](#). This function will cause a [Thread](#) to pause until its [resume\(\)](#) function is called. This function will cancel any sleep timer currently in effect.

Returns

true on success, false on failure.

4.3.3.6 bool Thread::resume ()

Resumes a paused or sleeping [Thread](#).

Returns

true on success, false on failure.

4.3.3.7 `bool Thread::sleep (unsigned long t)`

Puts the [Thread](#) to sleep for a certain number of seconds. If already running, the thread's [loop\(\)](#) function will be allowed to finish, but will not be called again until the timeout has expired, or the [resume\(\)](#) or [kill\(\)](#) function has been called.

Parameters

<code>t</code>	The number of seconds the Thread is to sleep for.
----------------	---

Returns

true on success, false on failure.

4.3.3.8 `bool Thread::sleep_micro (unsigned long t)`

Puts the [Thread](#) to sleep for a certain number of microseconds. If already running, the thread's [loop\(\)](#) function will be allowed to finish, but will not be called again until the timeout has expired, or the [resume\(\)](#) or [kill\(\)](#) function has been called.

Parameters

<code>t</code>	The number of microseconds the Thread is to sleep for
----------------	---

Returns

true on success, false on failure.

4.3.3.9 `bool Thread::sleep_milli (unsigned long t)`

Puts the [Thread](#) to sleep for a certain number of milliseconds. If already running, the thread's [loop\(\)](#) function will be allowed to finish, but will not be called again until the timeout has expired, or the [resume\(\)](#) or [kill\(\)](#) function has been called.

Parameters

<code>t</code>	The number of milliseconds the Thread is to sleep for.
----------------	--

Returns

true on success, false on failure.

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

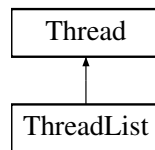
- [mthread.h](#)
- [mthread.cpp](#)

4.4 ThreadList Class Reference

An object for running several [Thread](#) objects simultaneously. A [ThreadList](#) object is a [Thread](#) in and of itself. This allows the creation of tiered [ThreadList](#) objects by placing a lower-priority [ThreadList](#) inside of a higher-priority [ThreadList](#).

```
#include <mthread.h>
```

Inheritance diagram for ThreadList:



Public Member Functions

- [ThreadList](#) (bool keep=false)
Constructor.
- [~ThreadList](#) ()
Destructor.
- bool [add_thread](#) ([Thread](#) *t)
Adds a [Thread](#) to the [ThreadList](#).

Protected Member Functions

- bool [loop](#) ()
The main loop.

Private Attributes

- [Thread](#) ** [thread](#)
An array of pointers to the [Thread](#) objects in the list.
- unsigned [thread_count](#)
The number of [Thread](#) objects in the list.
- unsigned [thread_index](#)
The index number of the active thread.
- bool [keep_flag](#)
If true, the [ThreadList](#) will not destroy itself when it becomes empty.

4.4.1 Detailed Description

An object for running several [Thread](#) objects simultaneously. A [ThreadList](#) object is a [Thread](#) in and of itself. This allows the creation of tiered [ThreadList](#) objects by placing a lower-priority [ThreadList](#) inside of a higher-priority [ThreadList](#).

Note

DO NOT place a [Thread](#) in more than one [ThreadList](#) or more than once in a single [ThreadList](#). DO NOT place a [ThreadList](#) inside of itself or one of its children. Also, DO NOT place the [main_thread_list](#) in another [ThreadList](#). These WILL cause memory corruption (and are silly things to do in the first place).

4.4.2 Constructor & Destructor Documentation

4.4.2.1 [ThreadList::ThreadList](#) ([bool](#) *keep* = *false*)

Constructor.

Parameters

<i>keep</i>	If true, the ThreadList will continue to run even after it's empty, otherwise it will automatically destroy itself once all of its Thread objects have finished.
-------------	--

4.4.3 Member Function Documentation

4.4.3.1 [bool](#) [ThreadList::add_thread](#) ([Thread](#) * *t*)

Adds a [Thread](#) to the [ThreadList](#).

Parameters

<i>t</i>	A pointer to the Thread to be added.
----------	--

Returns

true on success, false on failure.

4.4.3.2 [bool](#) [ThreadList::loop](#) ([void](#)) [[protected](#), [virtual](#)]

The main loop.

See also

[Thread::loop\(\)](#).

Reimplemented from [Thread](#).

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

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

Chapter 5

File Documentation

5.1 mthread.cpp File Reference

```
#include "mthread.h"
```

Functions

- void **loop** ()

Variables

- [ThreadList](#) * [main_thread_list](#) = new [ThreadList](#)
A pointer to the main [ThreadList](#). This object will be run in place of the loop function expected in most Arduino programs.

5.1.1 Detailed Description

Author

Jonathan Lamothe

5.2 mthread.h File Reference

```
#include "../newdel/newdel.h"
```

Classes

- class [Thread](#)
Provides thread functionality.

- class [ThreadList](#)
An object for running several [Thread](#) objects simultaneously. A [ThreadList](#) object is a [Thread](#) in and of itself. This allows the creation of tiered [ThreadList](#) objects by placing a lower-priority [ThreadList](#) inside of a higher-priority [ThreadList](#).
- class [EventHandler](#)
A [Thread](#) that only executes when a given event occurs.
- class [SwitchInput](#)
Handler for a switch input.

Defines

- #define [DEFAULT_DEBOUNCE](#) 50
Default switch debounce time.

Functions

- void **loop** (void)

Variables

- [ThreadList](#) * [main_thread_list](#)
A pointer to the main [ThreadList](#). This object will be run in place of the loop function expected in most Arduino programs.

5.2.1 Detailed Description

Author

Jonathan Lamothe