EmbeddedUtil

Release v0.3

Embedded Systems Department University Duisburg-Essen

Nov 20, 2020

FOR USERS

This is a collection of small utilities that can be used on resource constraint embedded systems.

CHAPTER

ONE

PERIODICSCHEDULER

1.1 EmbeddedUtilities/PeriodicScheduler.h

#include "EmbeddedUtilities/PeriodicScheduler.h"

Defines

PERIODIC_SCHEDULER_SIZE(*maximum_number_of_tasks*)

Typedefs

typedef enum *PeriodicSchedulerExceptions* **PeriodicSchedulerExceptions**

typedef uint16_t **Ticks**

typedef struct *Task* **Task**

typedef struct *PeriodicScheduler* **PeriodicScheduler**

Although the definition of this struct is visible further down, do not use the struct directly but use the functions to manipulate it. This way you ensure compatibility with upcoming changes.

typedef struct *InternalTask* **InternalTask**

Enums

enum PeriodicSchedulerExceptions

Values:

enumerator PERIODIC_SCHEDULER_FULL_EXCEPTION

enumerator PERIODIC_SCHEDULER_INVALID_TASK_EXCEPTION

Functions

void **processScheduledTasks**(*PeriodicScheduler* *self*)
Execute every task in the scheduler for which the configured time period has passed. After execution the period restarts.

void **updateScheduledTasks**(*PeriodicScheduler* *self*, *Ticks* *number_of_elapsed_ticks*)
Call this from your timer interrupt service routine. It updates all tasks in the Scheduler to reflect the ticks passed. This advances each tasks elapsed ticks by number_of_ticks.

size_t **getSchedulersRequiredMemorySize**(uint8_t *maximum_number_of_tasks*)
Returns the number of bytes needed for a Scheduler that can hold maximum_number_of_tasks.

uint8_t **addTaskToScheduler**(*PeriodicScheduler* *self*, **const** *Task* *task*)
Each task is copied to the internal array of tasks. You can therefore delete your task after this function returned. The operation additionally returns an id for the task, that can later be used to delete it. The id is guaranteed to be within 0 and the maximum number of tasks minus one. This fact can be used to save and manage data that needs be accessed during task excecution. Throws the PERIODIC_SCHEDULER_FULL_EXCEPTION when called while the number of free slots is zero.

uint8_t **scheduleTaskPeriodically**(*PeriodicScheduler* *self*, **const** *Task* *task*)
The same as addTaskToScheduler

void **removeAllTasksFromSchedule**(*PeriodicScheduler* *self*)
Removes all tasks from the current scheduler. This essentially frees all slots in the schedule, giving room for new tasks.

void **removeScheduledTask**(*PeriodicScheduler* *self*, uint8_t *id*)
Removes the task with the specified id from the schedule.

uint8_t **getNumberOfFreeSlotsInSchedule**(**const** *PeriodicScheduler* *self*)
Returns the remain free slots in the schdule. Use this function to determine how many tasks can still be added to the scheduler.

Task ***getScheduledTaskById**(**const** *PeriodicScheduler* *self*, uint8_t *index*)
Returns a pointer to the task with the specified id

PeriodicScheduler ***createPeriodicScheduler**(void *memory*, uint8_t *maximum_number_of_tasks*)
Creates a *PeriodicScheduler* struct at the given memory area. The memory area is assumed to be big enough to hold the *PeriodicScheduler* struct as well as the array of Tasks. You can use the Macro PERIODIC_SCHEDULER_SIZE to retrieve the necessary size at compile time. IMPORTANT: Before using the macro you will have to define PERIODIC_SCHEDULER_MAX_NUMBER_OF_TASKS

struct Task

Public Members

void (***function**)(void *argument)

void ***argument**

Ticks **ticks_elapsed**

Ticks **period**

struct InternalTask

Public Members

Task **task**

bool **is_valid**

struct PeriodicScheduler

Public Members

InternalTask ***tasks**

const uint8_t **limit**

1.2 File

```
#ifndef PERIODICSCHEDULER_PERIODICSCHEDULER_H
#define PERIODICSCHEDULER_PERIODICSCHEDULER_H

#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "CException.h"

/**
 * \file Util/PeriodicScheduler.h
 * Overall idea here is simple.
 * Have a timer interrupt set up, like so
 *
 * ```c
 * ISR(timer_vect)
 * {
 *  updateScheduledTasks(tasks, period);
 * }
 * ```
 *
 * and
 * ```c
 * void
 * updateScheduledTasks(PeriodicScheduler *self)
 * {
 *  // update period in ticks elapsed for all tasks
 * }
 * ```
 * then from main program call processTasks(Tasks *tasks)
 * implemented like so
 * ```c
 * void
 * processScheduledTasks(PeriodicScheduler *self)
 * {
 *  for (int i=0; i < number_of_tasks; i++)
 *  {
 *    if (taskIsDue(tasks[i]))
 *    {
 *      executeTask(tasks[i]);
```

(continues on next page)

(continued from previous page)

```
 *      resetTasksElapsedTicks(tasks[i]);
 *    }
 *  }
 * }
 * ```
 *
 * Most important the PeriodicScheduler is not coupled
 * to any clock by design. Instead it counts ticks.
 * The Scheduler will let all added tasks age by the specified number of ticks
 * for each call to updateScheduledTasks(). A call to
 * processScheduledTasks() will then go through all tasks
 * and execute each of them whose elapsed time is bigger
 * than the specified period.
 *
 * After execution the elapsed time is reset.
 * This means that tasks are not executed
 * at the point in time where the timer interrupt is issued,
 * but just on the next time the processScheduledTasks() function
 * is called. And the new period for these tasks restarts only then.
 * E.g. you want to call a function every 50ms, but you run your
 * processScheduledTasks only every 100ms then your task will get
 * executed only every 100ms. So if you need your tasks to be
 * executed in time you will have to make sure your processScheduledTasks()
 * function is executed fast enough and that each of your tasks
 * does not take as long as to block the next periods tasks.
 *
 * The library has debug statements built in. To enable them
 * you have to define the functions declared in Debug.h and
 * make sure they are linked into your executable before you link
 * against the PeriodicScheduler lib. We expect the debug functions to
 * not add a new line after a string.
 *
 */

typedef enum PeriodicSchedulerExceptions
{
  PERIODIC_SCHEDULER_FULL_EXCEPTION = 0x01,
  PERIODIC_SCHEDULER_INVALID_TASK_EXCEPTION,
} PeriodicSchedulerExceptions;

typedef uint16_t Ticks;

typedef struct Task
{
  void (*function)(void *argument);
  void *argument;
  Ticks ticks_elapsed;
  Ticks period;
} Task;

/**
 * Although the definition of this struct is visible
 * further down, do not use the struct directly but
 * use the functions to manipulate it. This way
 * you ensure compatibility with upcoming changes.
 */
typedef struct PeriodicScheduler PeriodicScheduler;
```

(continues on next page)

(continued from previous page)

```
/**
 * Execute every task in the scheduler for which
 * the configured time period has passed. After
 * execution the period restarts.
 */
void
processScheduledTasks(PeriodicScheduler *self);

/**
 * Call this from your timer interrupt service routine.
 * It updates all tasks in the Scheduler to reflect
 * the ticks passed.
 * This advances each tasks elapsed ticks by number_of_ticks.
 */
void
updateScheduledTasks(PeriodicScheduler *self,
                     Ticks number_of_elapsed_ticks);

/**
 * Returns the number of bytes needed for a Scheduler that
 * can hold maximum_number_of_tasks.
 */
size_t
getSchedulersRequiredMemorySize(uint8_t maximum_number_of_tasks);

/**
 * Each task is copied to the internal array of tasks.
 * You can therefore delete your task after this function returned.
 * The operation additionally returns an id for the task, that can later
 * be used to delete it.
 * The id is guaranteed to be within 0 and the maximum number of tasks minus one.
 * This fact can be used to save and manage data that needs be accessed during
 * task excecution.
 * Throws the PERIODIC_SCHEDULER_FULL_EXCEPTION when called while the
 * number of free slots is zero.
 */
uint8_t
addTaskToScheduler(PeriodicScheduler *self,
                   const Task        *task);

/**
 * The same as addTaskToScheduler
 */
uint8_t
scheduleTaskPeriodically(PeriodicScheduler *self,
                         const Task        *task);

/**
 * Removes all tasks from the current scheduler.
 * This essentially frees all slots in the schedule,
 * giving room for new tasks.
 */
void
removeAllTasksFromSchedule(PeriodicScheduler *self);

/**
```

(continues on next page)

(continued from previous page)

```
 * Removes the task with the specified id from
 * the schedule.  */
void
removeScheduledTask(PeriodicScheduler *self,
                    uint8_t id);

/**
 * Returns the remain free slots in the schdule.
 * Use this function to determine how many tasks can still
 * be added to the scheduler.
 */
uint8_t
getNumberOfFreeSlotsInSchedule(const PeriodicScheduler *self);

/**
 * Returns a pointer to the task with the specified id
 */
Task *
getScheduledTaskById(const PeriodicScheduler *self,
                     uint8_t index);

/**
 * Creates a PeriodicScheduler struct at the given
 * memory area. The memory area is assumed to be big
 * enough to hold the PeriodicScheduler struct as well
 * as the array of Tasks. You can use the Macro
 * PERIODIC_SCHEDULER_SIZE to retrieve the necessary
 * size at compile time. IMPORTANT: Before using
 * the macro you will have to define PERIODIC_SCHEDULER_MAX_NUMBER_OF_TASKS
 */
PeriodicScheduler *
createPeriodicScheduler(void   *memory,
                        uint8_t maximum_number_of_tasks);


#define PERIODIC_SCHEDULER_SIZE(maximum_number_of_tasks) ((( \
                                                    maximum_number_of_tasks)␣
↪\
                                                   * sizeof(InternalTask)) \
                                                  + sizeof( \
                                                    PeriodicScheduler))

typedef struct InternalTask
{
  Task task;
  bool is_valid;
} InternalTask;

struct PeriodicScheduler
{
  InternalTask *tasks;
  const uint8_t limit;
};

#endif //PERIODICSCHEDULER_PERIODICSCHEDULER_H
```

CHAPTER

TWO

DEBUG

Debug is a header only library. It provides macros that allow to create debug output, that can be completely removed via compiler optimization. The advantage lies in the fact that the compiler still sees the statements and one can be sure the functions are still used correctly even when debugging was not enabled for a long time.

The listed function declarations have to be implemented by the user.

2.1 How to use

Use the library by listing the `@EmbeddedUtilities//:Debug` target in the `deps` attribute of your `cc_*` rule.

Then include the header where needed:

```
#include "EmbeddedUtilities/Debug.h"
```

The header file provides the `debug` macro. Writing:

```
debug(MyType, variable);
```

will expand to:

```
printMyType(variable);
```

Additionally the header lists `printType(Type argument)` function declarations for commonly used types. As stated above Debug is a header only library. None of the function declarations ships with an implementation currently. You will have to implement every function yourself. There are plans to change this and implement efficient print functions.

To enable the compiler to remove content of unneeded debug output you should write the statements like this:

```
debug(String, "My debug output");
```

Do not use the debug function for Strings to print other things in a way like this:

```
char some_text[32] = {0};
sprintf(some_text, "My Message with int %i", some_int);
debug(String, some_text);
```

Doing this will in most cases lead to the array and the format string ending up in your program memory. Instead do the following:

```
debug(String, "My Message with int ");
debug(UInt8, some_int);
```

and implement both corresponding print functions.

To enable debug output compile the corresponding `*.c` files with the `-DDEBUG=1` flag. To disable the output specify `-DDEBUG=0` instead. Not setting the debug flag at all will result in a warning.

2.2 EmbeddedUtilities/Debug.h

#include "EmbeddedUtilities/Debug.h"

Implement print functions and compile with -DDEBUG=1 to have debug messages printed.

Defines

DEBUG

debug(*format_type*, *argument*)
This macro ensures that all calls to print functions are removed in the optimizing step if no debug messages are needed. Therefore, no implementation for those functions is required as long as you compile with DEBUG=0. The advantage of the macro is that the compiler still sees all the code when DEBUG is disabled and checks for correctness.

debugString(*message*)

debugUInt16(*message*)

debugNewLine(*message*)

debugDec16(*message*)

debugHex16(*message*)

debugDec8(*message*)

debugHex8(*message*)

debugHex32(*message*)

debugDec32(*message*)

debugDec32Signed(*message*)

debugChar(*message*)

debugBin8(*message*)

debugLine(*message*)

debugPtr(*message*)

Functions

void **printString**(**const** char*)

void **printChar**(char)

void **printHex32**(uint32_t)

void **printDec32**(uint32_t)

void **printDec32Signed**(int32_t)

void **printUInt16**(uint16_t)

void **printDec16**(uint16_t)

void **printHex16**(uint16_t)

void **printDec8**(uint8_t)

void **printBin8**(uint8_t)

void **printHex8**(uint8_t)

void **printPtr**(void*)

void **printNewLine**(void)

void **printLine**(**const** char*)

CHAPTER

THREE

MULTIREADERBUFFER

3.1 EmbeddedUtilities/MultiReaderBuffer.h

#include "EmbeddedUtilities/MultiReaderBuffer.h"

Defines

MULTI_READER_BUFFER_SIZE(*word_size*, *max_elements*, *max_readers*)

Expands to the number of bytes required to create a circular buffer with the provided parameters.

MULTI_READER_BUFFER_SIZE

This macro should be used to obtain the correct number of bytes to be allocated for creating a circular buffer. One buffer position is "wasted" for distinguishing whether the buffer is full or empty.

Parameters

- `word_size`: The size in byte of the individual data items that should be managed in the buffer
- `max_elements`: The maximum number of elements that the buffer should be able to hold at once
- `max_readers`: The maximum number of buffer readers that are allowed to exist in parallel

Typedefs

typedef struct *Buffer* **Buffer**

typedef struct *MultiReaderBuffer* **MultiReaderBuffer**

Enums

enum [anonymous]

Values:

enumerator BUFFER_UNDERRUN_EXCEPTION

More items than existent have been tried to be read by a reader.

enumerator BUFFER_OVERRUN_EXCEPTION

More items have been written than could have been read by a reader, i.e. unread items have been overwritten.

enumerator BUFFER_NO_FREE_READER_SLOTS_EXCEPTION

The number of reader slots is exhausted.

enumerator BUFFER_INVALID_READER_EXCEPTION
It has been tried to execute an operation with an invalid reader.

enum BufferReaderState
Indicates the different states a reader can have.

A reader is invalid as long as getNewBufferReaderDescriptor returns a descriptor for this particular reader to use for read operations.

Values:

enumerator BUFFER_READER_VALID
A reader is valid and can be used for read operations.

enumerator BUFFER_READER_OVERRUN
A valid read pointer has been overwritten by the write pointer, i.e. elements are lost.

enumerator BUFFER_READER_INVALID
A reader is invalid.

Functions

void **pushToBuffer**(*Buffer* **self*, **const** void **data*)
Writes new data into the buffer.

As with the nature of a circular buffer, the buffer space virtually never "ends" since when space is used up the buffer is written from the start again, overwriting the oldest entries.

Parameters

- `self`: A pointer to the circular buffer, addressed by its generalized type
- `data`: The data that should be written into the buffer

uint8_t **getNewBufferReaderDescriptor**(*Buffer* **self*)
Returns a descriptor for a new buffer reader.

The descriptor shall be used with functions deleteBufferReaderDescriptor, readableItemExistsForReader, popFromBufferWithReader and peekAtBufferWithReader.

Return A descriptor for the newly allocated reader, represented by an integer

Parameters

- `self`: A pointer to the circular buffer, addressed by its generalized type

Exceptions

- `BUFFER_NO_FREE_READER_SLOTS_EXCEPTION`: An exception is thrown when more readers have been tried to be allocated than are allowed; use deleteBufferReaderDescriptor to free unused readers.

void **deleteBufferReaderDescriptor**(*Buffer* **self*, uint8_t *reader_descriptor*)
Flags the provided reader descriptor as invalid.

Parameters

- `self`: A pointer to the circular buffer, addressed by its generalized type
- `reader_descriptor`: The descriptor of the reader with which the data should be retrieved

Exceptions

- `BUFFER_INVALID_READER_EXCEPTION`: An exception is thrown if the descriptor does not correspond to an actual reader slot

bool **readableItemExistsForReader**(**const** *Buffer* **self*, uint8_t *reader_descriptor*)

Returns whether there is at least one unread data item left that could be retrieved via popFromBufferWithReader or peekAtBufferWithReader.

Return true if the reader descriptor is valid and there are still elements left to be read; false if the reader descriptor is invalid or there are no elements left to be read

Parameters

- `self`: A pointer to the circular buffer, addressed by its generalized type
- `reader_descriptor`: The descriptor of the reader with which the data should be retrieved

const void ***popFromBufferWithReader**(*Buffer* **self*, uint8_t *reader_descriptor*)

Reads from the buffer including "removal" of the element that has been read.

The element is returned by reference, which needs to be casted into the actual data type (that the user must be aware of) and then dereferenced to retrieve the data item's value. The reader pointer will have proceeded by one data word after this function has been called.

If the reader pointer has been overrun by the write pointer, the reader pointer will be repositioned to the oldest entry in the buffer before an exception will be thrown. Therefore, a subsequent read operation will succeed, given no elements have been written in the meantime.

Return An "untyped" pointer to the current element the reader pointer is positioned at

Parameters

- `self`: A pointer to the circular buffer, addressed by its generalized type
- `reader_descriptor`: The descriptor of the reader with which the data should be retrieved

Exceptions

- `BUFFER_INVALID_READER_EXCEPTION`: An exception is thrown if the reader descriptor is invalid
- `BUFFER_OVERRUN_EXCEPTION`: An exception is thrown if the reader has been overrun by the write pointer, i.e. elements have been lost
- `BUFFER_UNDERRUN_EXCEPTION`: An exception is thrown if there are no elements left to be read for this reader, i.e. the buffer is "empty"

const void ***peekAtBufferWithReader**(**const** *Buffer* **self*, uint8_t *reader_descriptor*)

Reads from the buffer without "removing" the element that has been read.

The element is returned by reference, which needs to be casted into the actual data type (that the user must be aware of) and then dereferenced to retrieve the data item's value. The reader pointer will not have proceeded after this function has been called.

If the reader pointer has been overrun by the write pointer, the reader pointer will be repositioned to the oldest entry in the buffer before an exception will be thrown. Therefore, a subsequent read operation will succeed, given no elements have been written in the meantime.

Return An "untyped" pointer to the current element the reader pointer is positioned at

Parameters

- `self`: A pointer to the circular buffer, addressed by its generalized type

- `reader_descriptor`: The descriptor of the reader with which the data should be retrieved

Exceptions

- `BUFFER_INVALID_READER_EXCEPTION`: An exception is thrown if the reader descriptor is invalid
- `BUFFER_OVERRUN_EXCEPTION`: An exception is thrown if the reader has been overrun by the write pointer, i.e. elements have been lost
- `BUFFER_UNDERRUN_EXCEPTION`: An exception is thrown if there are no elements left to be read for this reader, i.e. the buffer is "empty"

void **initMultiReaderBuffer**(*MultiReaderBuffer* *self, uint8_t *word_size*, size_t *max_elements*, size_t *max_readers*)

Initializes a multi reader buffer.

This function creates all the necessary structures (see *MultiReaderBuffer*) to use the provided memory as a multi reader buffer and needs to be called first in order to use all the other functions provided by this header.

The memory passed via the first parameter needs to be large enough to hold the entire multi reader buffer given the set of customization parameters, i.e. the memory should be created using the MULTI_READER_BUFFER_SIZE with the same set of parameters.

Parameters

- `self`: Pointer to the memory that should be used for the circular buffer
- `word_size`: The size in byte of the individual data items that should be managed in the buffer
- `max_elements`: The maximum number of elements that the buffer should be able to hold at once
- `max_readers`: The maximum number of buffer readers that are allowed to exist in parallel

Variables

enum *[anonymous]* **MultiReaderBufferException**

struct MultiReaderBuffer

#include <MultiReaderBuffer.h> Defines the structure of the circular buffer implementation.

This circular buffer implementation allows to use multiple buffer readers, to provide a mechanism for implementing 1-to-n producer-consumer relationships in an efficient manner as often needed in the realm of embedded systems. It is utilized as part of the data processing pipeline implemented by the edge device.

Public Members

uint8_t **word_size_in_byte**
Stores the size in byte of the individual data items stored.

size_t **max_elements**
Stores the maximum number of elements.

size_t **max_readers**
Stores the maximum number of readers allowed in parallel.

void ***start**
Stores the pointer to the start position of the buffer.

void ***write**
Stores the writer pointer, which always points to the next position it would write to.

void ****readers**
: Holds the array of readers, where each reader is represented by a pointer into the buffer memory.

BufferReaderState ***reader_state_indicators**
: For each reader slot this array stores the current reader state (see BufferReaderState)

CHAPTER

FOUR

MUTEX

4.1 EmbeddedUtilities/Mutex.h

#include "EmbeddedUtilities/Mutex.h"

Typedefs

typedef struct *Mutex* **Mutex**

Functions

void **lockMutex** (*Mutex* **self*, void **lock*)

void **unlockMutex** (*Mutex* **self*, void **lock*)

void **initMutex** (*Mutex* **self*)

Variables

const uint8_t **MUTEX_WAS_NOT_LOCKED** = 0x01

const uint8_t **MUTEX_WAS_NOT_UNLOCKED** = 0x02

struct Mutex

Public Members

void ***lock**

4.1.1 File

```
#ifndef COMMUNICATIONMODULE_MUTEX_H
#define COMMUNICATIONMODULE_MUTEX_H

#include <stdint.h>

typedef struct Mutex Mutex;

struct Mutex
```

(continues on next page)

(continued from previous page)

```
{
  void *lock;
};

static const uint8_t MUTEX_WAS_NOT_LOCKED = 0x01;
static const uint8_t MUTEX_WAS_NOT_UNLOCKED = 0x02;

void
lockMutex(Mutex *self, void *lock);

void
unlockMutex(Mutex *self, void *lock);

void
initMutex(Mutex *self);

#endif //COMMUNICATIONMODULE_MUTEX_H
```

4.2 EmbeddedUtilities/Atomic.h

#include "EmbeddedUtilities/Atomic.h"

Functions

void **executeAtomically**(GenericCallback *callback*)

4.2.1 File

```
#ifndef COMMUNICATIONMODULE_ATOMIC_H
#define COMMUNICATIONMODULE_ATOMIC_H

#include "EmbeddedUtilities/Callback.h"

/**
 * \file Util/Atomic.h
 *
 * This function has to be implemented
 * by the user of the library.
 * The function provided as parameter
 * is assumed to be executed without
 * being interrupted. Enabling and
 * disabling interrupts usually depends
 * on the platform the application runs on.
 * To avoid introducing this dependency into
 * the communication module we rely on the
 * user providing this function during
 * link time. For most atmega
 * platforms an implementation
 * like the following should be sufficient:
 *
 * ```c
```

(continues on next page)

(continued from previous page)

```
 * #include <util/atomic.h>
 *
 * void
 * executeAtomically(GenericCallback callback)
 * {
 *   ATOMIC_BLOCK(ATOMIC_STATERESTORE)
 *   {
 *     callback.function(callback.argument);
 *   }
 * }
 * ```
 */

void
executeAtomically(GenericCallback callback);

#endif //COMMUNICATIONMODULE_ATOMIC_H
```

INDEX

Symbols

A

B

C

D

E

G

I

L

M

P

R

S

T

U