SaliMLib

cooperative Minimal Multitasking Library for 32-bit single-core Microcontrollers

version 0.2
document revision 1

Author Sibilev A.S.

SaliMLib

SaliLab cooperative Minimal Multitasking Library for 32-bit single-core Microcontrollers

SaliMLib is a library for use as a multitasking framework for embedded systems based on 32-bit single-core microcontrollers. This library uses cooperative multitasking. This means that each task is not allocated a pre-defined time quantum, but the task itself determines the appropriate time to transfer control to another task. This approach greatly simplifies task synchronization and eliminates the need to use critical sections.

The main disadvantages of cooperative multitasking compared to preemprtive multitasking are the uncertainty of the response time to events, which is why such systems are not used in real-time systems. However, this limitation can be easily circumvented by using hardware interrupts, thus obtaining a complete analog of "hard time systems", and the actual "real time" can be achieved by configuring the microcontroller peripherals to automatically work in parallel with the microcontroller core.

This library is an excellent alternative or addition to the traditional "state machines"for embedded systems. With this multitasking library, the implementation of many algorithms is radically simplified.

Features

- minimalism. This library consumes less resources than other RTOS
- multitasking. The library provides cooperative multitasking
- a two-level priority. In the library, all tasks are divided into critical and other tasks. Tasks with a critical priority level are managed first
- a model without priorities can be used
- no task scheduler. Based on the principle of cooperation all tasks of the same priority are performed sequentially
- computer language. It uses C++ 11 for the interface and platform-independent part, as well as Assembler language for the platform-dependent part
- simplicity. The entire library consists of three files: a kernel header file, a file with kernel source codes, and an assembler file with platform-dependent code
- documentation. The source codes of the library are provided with comprehensive comments, as well as a guide
- examples. There are usage examples for all parts of the library

Contents

Introduction

Minimalism

Traditional systems for microcontrollers are initially planned taking into account the minimum resources used. However, this library is one of the record holders for the minimum memory used (RAM and constant) and processor time.

Simplicity

The library's source code is contained in only three files: the header file, the platform-independent part, and the platform-dependent part. The core interface of the library has only about 10 functions, and a full study of its features will take a few minutes.

Scheduler

This library uses a two-level priority for task execution: critical tasks and others. Tasks with a critical priority level are managed first. You can also use a non-priority model, in which case all tasks must have a normal priority.
Tasks of the same priority are served sequentially.
Since in cooperative multitasking, each task itself determines when it is ready to give control to another task, there is no scheduler as such. Control is simply transferred to the next completed task, taking into account priorities.

Used language

The implementation of this library uses C++ 11 for the interface and platform-independent part, as well as assembly language for the platform-dependent part. Using C++ instead of traditional C is due to improved usability and some improvements in type control, which reduces the likelihood of errors.
The need for C++11 (or older) compared to earlier standards is due to the widespread use of lambda expressions.

Using SaliMLib

Using the library involves including its source code in the target project. Just copy two files of the platform-independent part: **SaliMCore.h** and SaliMCore.cpp to your project. For the platform-dependent part, select the file from the port directory that corresponds to your hardware and the compiler used.

Add the following line to the system tick timer handler:

```
smTickCount++; // Increment of the system tick counter
```

In the main function, you must also add system initialization

```
smInit(100);
```

where 100 is the stack size (in 32-bit words) for the default task.

For example arm cortex

```
void SysTick_Handler(void)
  {
```

```
  smTickCount++; // Increment of the system tick counter
  HAL_IncTick();
  }

void main(void)
  {
  //All initialisation
  ...
  //Initialisation of SaliMLib
  //Any of SmWaitXXX functions must be called AFTER smInit
  smInit(100);

  //Main loop
  while(1) {
    ...
    }
  }
```

For efficiency reasons, the SaliMLib library does not use dynamic memory allocation. It completely omits the new and delete operations. Therefore, the number of tasks in one project is fixed. This number is set by the SM_TASK_MAX global macro and is set to 8 tasks by default. To change this number, define the global macro SM_TASK_MAX with the required number of tasks, for example:

#define SM_TASK_MAX 10

By default, the use of namespaces is disabled in the SaliMLib library. Therefore, all library functions and classes are visible in the global namespace. Due to the fact that the library is included in the project may be a conflict of names. To avoid this, just enable the use of a namespace in the library. To do this, define the global SM_NAMESPACE macro with the desired namespace ID. For example:

#define SM_NAMESPACE MySpaceSaliMLib

A traditional example of the "Hello world" program for embedded systems is the led flashing program.

```
#include "SaliMCore.h"

extern "C" {

//System tick handler
void SysTick_Handler(void)
  {
  smTickCount++; // Increment of the system tick counter
  }

int main(void)
  {
  //All initialisation
  ...
  //Initialisation of SaliMLib
  smInit(100);

  //Main loop
  while(true) {
    //Led is on
    LedOn();
    //Wait for appropriate time
    smWaitTick(300); //Here is switch to execution of other task

    //Led is off
    LedOff();
    //Wait for appropriate time
    smWaitTick(300); //Here is switch to execution of other task
    }
  }
}
```

The structure of this program is identical to the traditional one. The main difference is that while waiting for a given interval, the traditional program does nothing and takes up CPU time in vain, while SaliMLib processes other system tasks during this time.

Task creation

Tasks are quasi-parallel functions. "Quasi" means that since the processor has one core, it performs one task at the same time, however, due to the rapid switching between tasks, the illusion of their parallel execution is created.

You can create a task using the smTaskCreate function:

```
void taskFunction( void* )
  {
  ...//Some execution with periodic call one of smWaitXxx functions
  }

void exampleOfTaskCreation()
  {
  //Create first task
  smTaskCreate( 300, nullptr, taskFunction );
  //Create second task
  smTaskCreate( 300, nullptr, taskFunction );
  //Some execution with call one of smWaitXxx functions
  ...
  }
```

In this example, taskFunction is a task function that will be executed in parallel with other tasks.

To create a task, call the smTaskCreate function and pass the task function to it. Multiple tasks can be created using the same task function. In any case, these will be two (or several) independent tasks. When creating a task, the size of the task stack in 32-bit words and an arbitrary parameter passed to the task function are also passed.

When creating a task, an arbitrary parameter is passed, which is then passed to the task function. For example:

```
void pinSet( int pin );
void pinReset( int pin );

void taskBlinkPin( int *pinIndex )
  {
  while(true) {
    //Set pin to 1
    pinSet( *pinIndex );
    //Wait time interval
    smWaitTick(300);
    //Reset pin to 0
    pinReset( *pinIndex );
    //Wait time interval
    smWaitTick(300);
    }
  }

void example() {
```

```
  int pin1 = 1;
  int pin2 = 2;
  smTaskCreateClass<int>( 300, &pin1, taskBlinkPin );
  smTaskCreateClass<int>( 300, &pin2, taskBlinkPin );
  }
```

Here, the taskBlinkPin task flashes an output whose number is passed to it as a parameter. And the actual output number for a specific task is specified when creating the task. For this purpose, the example uses the template of the smTaskCreateClass function, which converts the parameter type to the void type and calls smTaskCreate. If your task function uses a parameter, it is preferable to use a template before converting the type itself, since the template does this consistently, which ultimately reduces the likelihood of errors.

The issue of passing multiple parameters is solved by passing a pointer to a structure where multiple parameters can be described.

Task switching

Unlike systems with preemptive multitasking, where the moment when control is transferred to another task is determined by a timer, for systems with cooperative multitasking, the tasks themselves determine the time to switch. Therefore, in systems with preemptive multitasking, there are certain problems of synchronization and sharing, which are solved using special tools of the operating system. Considered example:

```
 //Stack
 mTopOfStack = smTopStack;
 smTopStack -= stackCellSize * 4;
```

where mTopOfStack is a local variable and smTopStack is a global variable. If this fragment is used by several tasks, keep in mind that switching tasks may occur at the moment between two operators. In this case, the tasks will get an incorrect smTopStack value, and the program execution will be disrupted, which will cause the system to crash. The main difficulty is that this effect may not occur for a long time and will appear after the system is shipped to the customer. In traditional programs and multitasking programs with cooperative multitasking, this effect occurs between the main program code and the interrupt code. Therefore, you need to pay special attention to sharing resources from the main code and from interrupts. In multitasking programs with push-out multitasking, this effect is present throughout the program, so you need to monitor access to shared resources absolutely throughout the program. This also applies to such common operations as allocating memory from the heap. In programs with push-out multitasking, allocating (and freeing) memory from the heap must be protected from task switching. Accordingly, standard libraries are not suitable and special wrappers should be used. For cooperative multitasking, no additional libraries are required and standard libraries can be freely used.

Thus, for a program with cooperative multitasking (including the SaliMLib library), if the global variable from the above fragment is not used in interrupts, then no additional measures need to be taken.

For systems with cooperative multitasking, the most important element of the system is a set of task switching functions. For the SaliMLib system, There is a single function that performs this switching:

```
 //!
 //! \brief smWaitVoid   Main wait function. Wait while waitFuncion return
true. While task is in wait state cpu switch to other tasks
 //! \param arg          Any argument for waitFuncion
```

```
 //! \param waitFunction Pointer to wait function
 //!
 void smWaitVoid( void *arg, SmWaitFunction waitFunction );
```

where waitFunction is a callback function that determines whether a given task is ready to continue execution. Until this function returns false, the task will not receive control and it will be distributed to other tasks. arg is an arbitrary parameter that is passed from the task to the callback function. This pointer can be to any objects: global, local, temporary, from the heap.

As soon as the waitFunction function returns true, the task gets control.

When calling smWaitVoid, the system will try to transfer control to another task anyway. If there are no tasks ready for execution and the wait Function returns true, control returns to the current task.

Two template wait functions are used to automatically convert parameters, implement additional type control, and eventually use smWaitVoid anyway.

Conversion function for using a pointer to an arbitrary object.

```
 //!
 //! \brief smWait       Template for automatic conversion of waitFunction
argument. It simply converts specified pointer to void
 //!                     for smWait and waitFunction
 //! \param arg          WaitFunction argument
 //! \param waitFunction Pointer to wait function
 //!
 template <class SmArg>
 void smWait( SmArg *arg, bool (*waitFunction)( SmArg *farg ) )
   {
   smWaitVoid( arg, (SmWaitFunction)(waitFunction) );
   }
```

Function-transformation for using an object with a specified "function call" operation.

```
 //!
 //! \brief SmWaitClass Template for automatic using class operator() member-
function for stop wait event.
 //!                    StClass must have member-function with prototype "bool
operator () () { ... }".
 //!                    When this function return true waiting stop and task
resumed.
 //! \param cls         Any class object which has member-function operator().
 //!
 template <class SmClass>
 void smWaitClass( SmClass cls )
   {
   smWait<SmClass>( &cls, [] ( SmClass *ptr ) -> bool { return ptr(); } );
   }
```

Using C++ makes it possible to use lambda functions as callback functions:

```
void SaliCom::smYeld()
 {
 smWaitVoid( nullptr, [] ( void* ) -> bool { return true; } );
 }
```

This technology improves the structure of the code, increases readability, and reduces the likelihood of errors.
It was decided to abandon the use of closures for efficiency reasons. If you apply closures, this entails automatic use of libraries, as well as new and delete operations. In addition, the volume of compiled code is growing significantly.

For convenience, several auxiliary waiting functions are provided. These are inline functions and in fact only perform type conversions and validation.

```
 //!
 //! \brief smWaitBoolTrue Helper function which wait until bool is set to true
 //! \param arg            Testing bool
 //!
 inline void smWaitBoolTrue( bool *arg );

 //!
 //! \brief smWaitBoolFalse Helper function which wait until bool is set to
false
 //! \param arg             Testing bool
 //!
 inline void smWaitBoolFalse( bool *arg );

 //!
 //! \brief smWaitIntUntilZero Helper function which wait until int is set to 0
 //! \param arg                Testing int
 //!
 inline void smWaitIntUntilZero( int *arg );

 //!
 //! \brief smWaitIntUntilNotZero Helper function which wait until int is set
to not 0
 //! \param arg                   Testing int
 //!
 inline void smWaitIntUntilNotZero( int *arg );
```

Convenience of multitasking programming

Using multitasking, and in particular cooperative multitasking, makes it easier to write embedded programs, radically improving its readability.

For example, the algorithm for writing a sector to external flash memory:

```
//Check flash-memory ready
//Sector erase
//Sector write
```

Each stage of this algorithm involves data exchange with flash memory (possibly more than one), for example, via spi. And each such exchange implies the following algorithm:

```
//1.Activate CS
//2.Start DMA or interrupt interchange
//3.Wait for interchange completion
//4.Deactivate CS
```

Unfortunately, step 3 of the algorithm is a wait and a waste of CPU time. Often this step looks like this:

```
while( !isTransferComplete() );
```

To take full advantage of this time in traditional programs is a very time consuming task. In fact, we have to implement this algorithm as a step-by-step state machine. This approach is extremely unclear and fraught with errors. Using the SaliMLib multitasking library allows you to maintain the visibility of a traditional program and avoid losing CPU time. The only modification you need to make to the code is to replace the wait line with the following one:

```
smWaitVoid( nullptr, [] (void*) -> bool { return isTransferComplete(); } );
```

This ensures that control is switched to other tasks until the exchange is completed. When working with peripherals, interrupt access or DMA is often used. Using these technologies gives the best effect when using SaliMLib, while maintaining the visibility of sequential algorithms.

Working with time

Usually, to implement work with time, use the decrement of variables:
A global variable is defined:

```
//Some global variable
int someTimeOut;
```

The system timer interrupt handler is added to:

```
//Timer decrement in system timer handler
if( someTimeOut ) someTimeOut--;
```

Then the program initializes the timer:

```
//Timer setup
someTimeOut = 1000;
```

Then it is periodically checked and some actions are performed after it expires (or until it has expired) :

```
if( someTimeOut == 0 ) {
  //Time expired, do something
  }
```

If there are several timers, all the specified steps are repeated. The program size and resources consumed grow linearly with the number of timers. This approach is simple, but inefficient.
Instead, SaliMLib uses a different principle:
There is a single global tick counter

```
volatile int smTickCount;
```

The system timer interrupt handler is added to:

```
//Tick counter increment
smTickCount++;
```

Each timer in SaliMLib is a simple integer variable that represents a point in time in the future. By comparing this value of this variable with the clock counter, you can determine when this moment

will occur. Therefore, only the system tick counter is incremented, not every timer. This approach drastically reduces the computational load of maintaining each timer and simplifies the code.

The smTickFuture function is used to get this point in the future relative to the current value of the system clock counter. For example:

```
int localTimer = smTickFuture( 1000 );
```

Creates a moment in the future that is 1000 clock cycles away from the current one. Accordingly, the timer will be triggered 1000 cycles after this moment.

Next, if you need to create a timer, it can be created locally:

```
void f()
 {
 //Timer create
 int localTimer = smTickFuture( 1000 );
 //That equals to localTimer = smTickCount + 1000;

 if( smTickIsOut(localTimer) ) {
   //Time expired, do something
   }
 }
```

The smTickIsOut function compares the timer value with the tick counter. as soon as they are equal, this function returns true. The duration of the system operation is not limited by the capacity of the tick counter variable. Counting time intervals works fine even after this counter is" wrapped " (overflowed). However, there is a limitation. It consists in the maximum duration of the counted intervals. It is about 2 billion ticks. If the tick counter increases every millisecond (the normal value), the maximum time interval is about 23 days. This limitation is insignificant for the vast majority of embedded systems and can be solved for counting long intervals by using, for example, the built-in calendar.

This timer system can be used without SaliMLib.

With SaliMLib system timers adds the functionality of the timing and functions of expectations. For example, using the standby function to set the led to blink:

```
 while(true) {
   //Led on
   LedOn();
   //Wait for 300 ticks
   smWaitTick(300); //Here is switch to execution of other task
   //Led off
   LedOff();
   //Wait for 300 ticks
   smWaitTick(300); // Here is switch to execution of other task
   }
```

Timers can also be used to limit the waiting time for events:

```
void f()
 {
 //Create local timer
 int localTimer = smFutureTime( 1000 );

 //Wait for operation complete or time out
 smWait<int>( &localTimer, [] ( int *localTimerPtr ) -> bool {
```

```
   return smTickIsOut(*localTimerPtr) || isOperationComplete();
   });

 if( smTickIsOut(localTimer) ) {
   //Time out
   }
 else {
   //Operation completed successfully
   }
 }
```

Auxiliary synchronization elements

Mutex object

Usually, mutexes are used to block access to a specific resource by other tasks while the resource is occupied by the current task. In systems with push multitasking, mutexes are system objects provided by the operating system. This complexity is due to the need for atomicity of operations with mutexes and their direct impact on the internal queues and lists of the operating system.

In the SaliMLib library, a regular logical variable "associated" with a resource can act as a mutex. For example, consider an SPI port that can access multiple external devices. In this case, the mutex can be organized as follows:

```
//SPI structure from ST library
SPI_HandleTypeDef hspi3;

//Variable which acts as SPI3 port locker
bool              spi3Busy;

//Function to lock resurs
void spi3Lock()
  {
  if( spi3Busy )
    smWaitBoolFalse( &spi3Busy );
  spi3Busy = true;
  }

//Function to unlock resusr
inline void spi3UnLock() { spi3Busy = false; }

void f()
  {
  //Lock resurs
  spi3Lock();

  //Do something

  //Unlock resource
  spi3UnLock();
  }
```

To support mutexes, SaliMLib has The SmMutex helper class. It is built on a boolean variable and its internal implementation is identical to the one presented above.

```
 //!
 //! \brief The SmMutex class Helper class for guard some resource against
sharing
 //!
 class SmMutex {
     bool mBusy; //! Variable to indicate resource is busy
   public:
     //!
     //! \brief SmMutex Construct initialy not busy resource
     //!
     SmMutex() : mBusy(false) {}

     //!
     //! \brief isLocked Check if resource is busy
     //! \return         true when resource is busy
     //!
     bool isLocked() const { return mBusy; }

     //!
     //! \brief lock Try lock resource. If resource is busy then it wait until
it will be free.
     //!             If resource is free it locked
     //!
     void lock() {
       if( mBusy )
         smWaitBoolFalse( &mBusy );
       mBusy = true;
       }

     //!
     //! \brief unlock Unlocks resource
     //!
     void unlock() { mBusy = false; }
   };
```

Using the SmMutex class, the above snippet will look like this:

```
//SPI structure from ST library
SPI_HandleTypeDef hspi3;
//Mutex which acts as SPI3 port locker
SmMutex  spi3Mutex;

void f()
  {
  //Lock resource
  spi3Mutex.lock();

  //Do something

  //Unlock resusr
  spi3Mutex.unlock();
  }
```

In order not to get confused with the number of locks and unlocks, the number of which must be exactly the same, SaliMLib provides the traditional SmMutexLocker helper class in such cases.

This object is used as a local variable of the function. When an object of this class is created, the resource is automatically blocked, and when the object is automatically deleted, the resource is automatically released.

```
//SPI structure from ST library
SPI_HandleTypeDef hspi3;
//Mutex which acts as SPI3 port locker
SmMutex  spi3Mutex;

void f()
  {
  //Lock resource
  SmMutexLocker locker(spi3Mutex);

  //Do something

  //When returning from the function, the locker object will be automatically
  //deleted, and the resource will be unlocked
  }
```

Semaphores

Some resources have multiple but limited access. In this case, semaphores are usually used, which work similarly to mutex, but still support the counter of available resources. The SaliMLib system provides two classes for working with semaphores: SmSemaphor and SmSemaphorLocker. Their functionality is similar to the corresponding SmMutex classes.
Although the library has a semaphore implementation, its use in programs is extremely insignificant, due to the possibility of direct control of resources. For example, a semaphore may reflect the number of available items in the queue. In this case, adding elements will look like this:

```
using QueueChar100 = SmFixedQueue<char,100>;
QueueChar100 queue;

void f()
  {
  //Let's say we want to add an item to the queue.
  //First we check whether there is free space in the queue
  if( queue.emptyCount() == 0 )
    //There are no free places, we are waiting for them to appear
    smWait<QueueChar100>( &queue, [] ( QueueChar100 *q ) -> bool
{ return q->emptyCount() != 0; } );
  //The item to be added
  queue.enque( 'A' );
  }
```

This example shows a queue from SaliMLib, and the specified algorithm will work fine for any other implementations, including standard libraries.

Fixed-size containers

For embedded systems, they often try to exclude the use of the heap. This is due to the unpredictable behavior of the memory allocation system in conditions of limited memory. For example, the amount of memory may be sufficient, but due to fragmentation, the memory allocation operation may fail. Therefore, SaliMLib offers several templates for organizing standard fixed-size containers. A special feature of the implementation is that waiting functions are already built into the add and extract operations (free space is expected for add operations, and elements are expected for extract operations).

Fixed-size queue

The most widely used container in embedded systems is a queue. Support for queues in SaliMLib is carried out using the template SmFixedQueue. In addition to traditional extraction and deletion operations, the queue provides access to the maximum contiguous section. This access is convenient for implementing algorithms for sending data blocks.

```
//UART structure from ST library
UART_HandleTypeDef huart3;

using QueueChar1000 = SmFixedQueue<char,1000>;

QueueChar1000 uartQueue;

//The task of transmitting characters from the queue in blocks using DMA
void uartSenderTask( void* )
  {
  while(true) {
    //Waiting for characters in the queue
    uartQueue.waitContinueItem();
    int count = uartQueue.continueCount();

    //Initiate a dma exchange
    HAL_UART_Transmit_DMA( &huart3, (uint8_t*) uartQueue.continueBuffer(),
count );

    //Wait for the exchange to complete
    smWait<UART_HandleTypeDef>( &huart3, [] ( UART_HandleTypeDef *huart ) ->
bool { return HAL_UART_GetState(huart) == HAL_UART_STATE_READY; });

    //Exchange completed, remove the transferred characters from the queue
    uartQueue.continueDeque(count);
    }
  }
```

The queue provides quick insertion into the tail and quick removal from the head. This time is fixed and does not depend on the queue size. The queue provides constant time for accessing any items located in the queue.

A queue can also work for communication between the interrupt handler and the main program. For example, if a character is interrupted by uart, it can be added to the queue, and extracted and analyzed in the main queue.
Important! When Queuing in interrupts, you must manually check the availability of space in the queue to avoid switching tasks built into these functions.

Fixed-size stack

The stack is represented by the SmFixedStack template.
You cannot use the stack in interrupts.
The stack provides fast insertion and extraction from the stack. This time is fixed and does not depend on the queue size. The stack provides constant time for accessing any elements located on the stack.

Fixed-size buffer

Another container class is SmFixedBuffer. A buffer is an array with specific add, insert, and delete operations. The add operation takes constant time and depends on the size of the data, but does not depend on the size of the buffer. Insert and delete operations move data located in the buffer, so their operation time depends linearly on the distance to the end of the buffer. The buffer provides constant time for accessing any elements located in the buffer. In addition, the buffer provides continuity of the data placement array.