FabOS

a.k.a.: Fabian's OS

Programmers guide

www.huslik.net

Content

1. What is FabOS

FabOS is a lightweight embedded operating system aimed for small processors and is completely written in C using as less RAM and code space as possible.
It is -as far as possible- independent of the compiler and processor used, therefore it is quite easy to port it to another processor type or compiler.

Footprint:
For a AVR (MEGA32) sample implementation with 3 Tasks (4 including idle), 3 alarms and 3 mutexes, it takes 38 to 51 bytes RAM and 1774 to 2196 bytes code, depending on features used.
The values are net values. The stack space for every task and possibly used queue space is to be added.

Key features:
- Up to 64 tasks
- Preemtive real time task scheduling
- Low interrupt blocking by scheduling
- Mutexes with priority inversion
- Up to 8 different events per task
- User defined Queues with flexible element size
- Several timed Alarms per Task
- Flexible OS-tick timer interrupt - any interrupt source can be used for scheduling
- The scheduling interrupt remains usable for the application
- Idle task, which runs on "native" stack
- Optional run-time checks
- Easy to use and to understand

Scope of delivery:
- Example AVR-Studio project
- main.c example with three separate “task” files
- confguration example FabOS_config.h
- The operating system core files FabOS.c and FabOS.h

In case of a royalty free license, you may only obtain an object code file instead of the FabOS core files. Please specify the processor and FabOS_config.h settings, you want to use, when ordering your free copy..
For compiling a project with an object file, refer to 15. Building FabOS with an object file on page 23.

2. License Agreement

3. Concepts

This is a static operating system.
The switching of the tasks is done in a non-cooperative way, so called pre-emptive task switching.
One timer interrupt is needed for scheduling.
The task switch is done by storing the context (all registers + CPU state) on top of the stack of the task and restoring the context of the other task.
After the safe and restore operation, the processing of the task is continued, as if nothing happened.
The interrupt stack resides on top of the stack of the actual task.
There is an idle task, which is executed, if no other task is ready to run.

A System consists of some tasks, which are declared as a “void” function with the following structure:

```
void myTask(void)
{
    while(1)
    {
     … your code …
    }
}
```

Each if this task-functions consists of an eternal loop with your instructions.
The switching between the tasks is then done pre-emptively by the task-switching interrupt.
Each task has its own stack, provided by a globally declared byte array.
The idle task does not have a stack-array declared. Instead it uses the unused "ordinary" stack at the end of RAM. The idle task consists of an infinite loop placed after the call of OS_StartExecution().

Summary of some key facts:

- This is a static operating system.
- An OS element once created, exists forever.
- The use of malloc() is not preferred in a static environment.
- One task has a fixed priority, which is at the same time the task ID. Lower IDs come first.
- Queues are static in length.

There are some macros defined to ease the configuration.
Please take a look at the example code for clarification.

4. Set up of the operating system

The set-up of the operating system is done at two points.
- the FabOS_config.h file and
- the main.c entry file of your application.

4.1. User configuration in FabOS_config.h

`#define OS_NUMTASKS 3`
Set the total number of tasks excluding the idle task. (number of OS_CreateTask – function calls)
The maximum number is 64. The maximum performance is reached with number <= 8.

`#define OS_NUMMUTEX 3`
Set the total number of different mutexes.
The maximum number is limited to 255.

`#define OS_NUMALARMS 5`
Set the total number of different Alarms.
The maximum number is limited to 255.

`#define OS_ScheduleISR TIMER1_COMPA_vect`
`#define OS_ALLOWSCHEDULING`
`#define OS_PREVENTSCHEDULING`
see section 4.2 Scheduling Timer configuration

`#define OS_USECLOCK 1`
Enable the use of a 32-bit OS tick counter, which can be read by OS_GetTicks(uint32_t* pTime)

`#define OS_USECOMBINED 1`
Use "OS_WaitTicks()" and "OS_WaitEventTimeout()" which is easier to use, than combining alarms and events to get the functionality.

`#define OS_USEEXTCHECKS 1`
Prevent false API usage (e.g. waiting in idle task)

`#define OS_USEMEMCHECKS 1`
Enable the OS_get_unused_Stack function.

`#define OS_UNUSEDMASK 0xAA`
Set the value to be written into unused memory in Task ID 0. Task 1 will be 0xAB and so on.
This may be useful to actually see the free stack space in your debugger.
Used by "OS_get_unused_Stack()"

`#define OS_TRACE_ON 1`
Enable trace to OS_Tracebuffer[]. The number of the OS_TRACE(n) macro is written into the OS_Tracebuffer array, when executing the OS-Code.
The actual(next) buffer is found in the OS_TracebufferIdx variable.
Setting this define to 0 fully disables the tracing and therefore needs no additional resources.

`#define OS_TRACESIZE 1000`
Size of OS_Tracebuffer[] (in bytes)

```
#define OS_TypeAlarmTick_t uint16_t
```
Change this type to uint32_t, if you need longer wait time than 65535 OS ticks.

4.2. Scheduling Timer configuration

In this section the configuration of the scheduling timer interrupt is explained:

In the FabOS_config.h the scheduling timer is selected:
```
#define OS_ScheduleISR INTERRUPT_VECTOR_NAME
```
The user must set up the cyclic timer interrupt within the CPU_Init() function.
If any additional code has to be executed in this interrupt (e,g, resetting the timer interrupt), the function "OS_CustomISRCode() " (see4.4 void OS_CustomISRCode())can be filled with this code. This function will be called in the interrupt context. Any other valid interrupt of the processor can be used for scheduling.

With the macros
```
#define OS_ALLOWSCHEDULING
#define OS_PREVENTSCHEDULING
```
the user must provide commands to enable or disable the selected timer interrupt.

The used timer interrupt is to be defined in FabOS_config.h for example (ATMEGA32):
```
#define OS_ScheduleISR            TIMER1_COMPA_vect
#define OS_ALLOWSCHEDULING        TIMSK |= (1<<OCIE1A);
#define OS_PREVENTSCHEDULING      TIMSK &= ~(1<<OCIE1A);
```

In the CPUinit() the used timer interrupt is to be set up with global interrupts **disabled**.
An useful frequency for the ISR may be 1 to 10 ms, depending on the application.
The user (you) is in charge to make the interrupt happen in the wanted OS cycle time.
If the Interrupt is activated before OS_StartExecution(), the OS will be corrupted.

4.3. OS_DeclareTask(TaskName, StackSize)

This is actually not a function, but a macro for the task function prototype definition, as well as the stack allocation. It is to be used before the main() function, close to where the global variables are declared. The task name is then used as the name of the void function, which declares the task. For the stack, the prefix “OSStack” is pre-pended inside the macro.

Stack size is given in bytes for each task. The space must be dimensioned at least 34 bytes larger than calculated for the program flow, because the task context is stored there at a task change. Provide extra stack space on top for interrupts, because every interrupt may occur within every task.

The remaining unused stack size can be checked during runtime by the OS_get_unused_stack(TaskID) function.

An example may look like:
```
OS_DeclareTask(myTask,200);
```
and expands as follows:
```
void MyTask(void); // function prototype
uint8_t OSStackMyTask[200]; // task stack definition
```

4.4. void OS_CustomISRCode()

This function is to be provided in the users code, if something has to be done in the timer-tick-

ISR. This function will be executed inside the ISR context. A good example is to set a compare register to some value to make the next timer interrupt occur at the correct time. The following is only an example.

```
void OS_CustomISRCode(void)
{
      // TODO add your Timer ISR here
      TCNT1 =0;  // reset the timer on ISR
}
```

4.5. void CPU_init()

In this function, which is provided by you, initialize the CPU and the timer interrupts including the cyclic task scheduling interrupt, but do not yet enable the global interrupts. This would confuse the scheduler.

4.6. void OS_CreateTask(TaskName, uint8_t taskNum)

Creates a task inside the main function. This function is realized as a macro to simplify the API; First Parameter is a pointer to a void function, which implements the task. Inside this function all OS – API calls can be used.
Second parameter is the task number, where also the task number defines the tasks priority. Task 0 has the highest priority. Task 7 is the lowest priority. The maximum number of tasks is OS_NUMTASKS. (so the highest number here is OS_NUMTASKS - 1)
In a bigger application it may be useful to use macros for the task IDs.

An example might look like:

```
OS_CreateTask( myTask, 1);
```

and expands to:

```
OS_TaskCreateInt(myTask, 1, StackmyTask , sizeof(StackmyTask));
```

which is the internal task creation function, which joins the task function and the task stack.

4.7. void OS_StartExecution()

Start the operating system; All tasks are executed at least once after start. This function returns, if no task is to be run for the first time. The code after this function call is the idle task. So this function must be followed by an infinite loop. If the following loop exits, there will be an uncontrolled reset of the processor or other strange behaviour.

5. The main()

The main() function may look like the following code:

The main function itself:

```
int main(void)
{
```

Initialisation of the CPU with global interrupts disabled afterwards:

```
      CPU_init();
```

The creation of the tasks (up to NUMTASKS) with priority, where 0 is highest prio;
The same priority (also Task ID) must only appear once.

```
      #define TSKID0 0
      #define TSKID1 1
      #define TSKID2 2 // in header-file!

      OS_CreateTask(Task1, TSKID0);
      OS_CreateTask(Task2, TSKID1);
      OS_CreateTask(Task3, TSKID2);
```

The creation of the alarms (up to OS_NUMALARMS) and linking them to Tasks via the Task ID:
One alarm is connected to a task, which it will wake up, if expired.
Several alarm may be connected to the same task.

```
      #define ALMID_A 0
      #define ALMID_B 1
      #define ALMID_C 2 // in header-file!

      OS_CreateAlarm(ALMID_A,TSKID0);
      OS_CreateAlarm(ALMID_B,TSKID0); // two alarms are bound to task 0
      OS_CreateAlarm(ALMID_C,TSKID1);
```

Starting of the Operating System:

```
      OS_StartExecution() ;
      while(1)
      {
            // THIS IS the idle task, which must never exit.
      }
}
```

If the main-function exits by accident, the CPU will reset or some other unpredictable results will occur. Do not enable interrupts globally during init. This is done by the call of OS_StartExecution().

6. Preemptive tasks

A System also contains some tasks, which are declared as a “void” function with the following structure:

```
void Task(void) { … your code ...}
```

A Task must never exit the function. Therefore the following is the minimum for a task.

```
void Task(void)
{
    while(1)
    {
          // add your code here...
    }
    // this line must never be hit!
}
```

If the function exits by accident, the CPU will reset or some other unpredictable results will occur. (e.g. infinite loop with no scheduling active)
A high priority task will run as long it does not wait for something (mutex, event, alarm etc..)

The prototype and definition of the task stack via a byte array, is declared by using the macro:
OS_DeclareTask(). (see 4.3 OS_DeclareTask(TaskName, StackSize))

6.1. Idle task

This is the code, which comes right after OS_StartExecution() in main().
Inside the idle task, it is not allowed to exit the function (so exiting main()) and it is also not allowed to use OS_Wait functions.
Which means, the idle task can only use:

```
void  OS_SetEvent(uint8_t EventMask, uint8_t TaskID);
void  OS_MutexGet(int8_t mutexID);
void  OS_MutexRelease(int8_t mutexID);
void  OS_SetAlarm(uint8_t AlarmID, uint16_t numTicks );
uint8_t    OS_QueueIn(OS_Queue_t* pQueue , uint8_t byte);
uint8_t    OS_QueueOut(OS_Queue_t* pQueue, uint8_t *pByte);
uint16_t   OS_GetUnusedStack (uint8_t TaskID);
void  OS_GetTicks(uint32_t* pTime);
```

7. Mutexes

7.1. void OS_MutexGet(int8_t mutexNum);

Get a mutex when entering a critical section. If the mutex is occupied, the scheduling will switch to the task, which is blocking the mutex. Otherwise the mutex will be locked and the execution continues right away.
For very short code in between the mutex get and release, it might be useful to use OS_PREVENTSCHEDULING and OS_ALLOWSCHEDULING to perform the mutex. This gives a much faster behaviour, because no task switch occurs.
The total number of mutexes is limited to NUMMUTEX.
Trying to get the same mutex twice without releasing it, leads to an eternal loop.
Triggers re-scheduling.

7.2. void OS_MutexRelease(int8_t mutexNum);

Release the mutex at the end of a critical section. A re-scheduling will be done, if another task is waiting for the mutex.
Triggers re-scheduling.

8. Alarms

Alarms wake up tasks after a certain time. An Alarm is always assigned to a task.
Alarms have to be pre-configured:

Use the
`OS_CreateAlarm(ALARMID, TASKID)`
macro to assign an alarm to a task before using it.
An alarm can always be only assigned to a single task, which waits for its expiry.
But more than one alarm can be assigned to one task.

8.1. void OS_WaitTicks(uint8_t AlarmID, uint16_t numTicks) ;

Inactivate Task and wait for the specified number of OS ticks.
Triggers re-scheduling. This is a macro, which uses OS_SetAlarm and OS_WaitAlarm internally.

8.2. void OS_SetAlarm(uint8_t AlarmID, uint16_t numTicks);

Set Alarm for own or other task and continue. An alarm timeout will be registered for the specified Alarm. To remove (=cancel) an alarm, set it again with numTicks = 0;

8.3. void OS_WaitAlarm(uint8_t AlarmID,);

Inactivate Task and wait for an alarm to expire.
Triggers re-scheduling.

9. Events

Events are “Triggers” to cause some other task to do something.
OS events are not to be confused with the XMEGA event system, which can of course be used in parallel.

9.1. void OS_SetEvent(uint8_t TaskID, uint8_t EventMask)

Set one or more events (8 bit EventMask) for one task.
Each task has up to 8 (built in) events, which are realized by a bitmask.
You can send one or more events to another task. It will wake up, if it waits for at least one of the events. If you send events, and the task is not waiting for it, it will just continue and be notified, if it later waits for it. (Events are never lost)
Triggers re-scheduling.

9.2. uint8_t OS_WaitEvent(uint8_t EventMask)

Inactivate Task and wait for one or more events. Return value is the bit-mask of the event(s), which lead to task reactivation. If the return value is zero, the task was activated by an expired alarm. It is unfortunately not possible to see, by which alarm, if more than one alarm is bound to the task.
Triggers re-scheduling.

To do a waitEvent with timeout, you can do:

```
SetAlarm(MyAlarm, MyTimeout);
ret = WaitEvent(myEventMask);
```

if (ret == 0) , then you had a timeout.
You have to disable the alarm in case of event success using:

```
SetAlarm(MyAlarm, 0);
```

...or just use the following simplified OS_WaitEventTimeout function.

9.3. uint8_t OS_WaitEventTimeout(uint8_t EventMask, uint8_t AlarmID, uint16_t numTicks)

Inactivate Task and wait for an event in combination with a timeout.
Continues, if:

- the timeout has expired (return value is 0)
- if an event had occurred (return value is the event, which lead to activation)

Triggers re-scheduling.

In this function the alarm will be changed. This means, it overwrites a possibly running alarm.

10. Queues

The queue memory must be globally declared and initialized as follows:
```
OS_DeclareQueue(QueueName, NumberOfElements, ElementSize);
```
It is to be used outside of main(), where global variables are declared.
For example:
```
OS_DeclareQueue(myQueue,100,4);
```
declares a Queue named "myQueue" which has room for 100 elements of a size of 4 bytes. Pay attention: the Queue mechanism reserves one element more than declared for empty/full recognition.

Technical information:
The macro
```
OS_DeclareQueue(myQueue,100,4);
```
expands to:
```
uint8_t OSQDmyQueue[101*4]; // OS Queue Data
OS_Queue_t myQueue = {0,0,4,101*4,OSQDmyQueue} // struct for management
```

That means, it declares an array for the queue data and a queue struct, which contains the information and pointers to the data array.

10.1. uint8_t OS_QueueIn(OS_Queue_t pQueue , uint8_t *pByte)*

Put an element into a FIFO queue.
Return value is 1 if queue is full, 0 if OK.

A 32bit-value can be stored to a queue as follows, if the queue is declared as above.
```
ret = OS_QueueIn(&myQueue,(uint8_t*)&my32BitValue);
```

10.2. uint8_t OS_QueueOut(OS_Queue_t pQueue, uint8_t *pByte)*

Get an element out of a FIFO queue.
Return value is 1 if queue was empty and no data was copied, 0 if OK.
Element size depends on the queue declaration.
The Pointer target must be of the same size as the source.

```
ret= OS_QueueOut(&myQueue,(uint8_t*)&my32BitValue);
```

11. Internal functions (normally not used by the user)

11.1. ISR (OS_ScheduleISR)__attribute__ ((naked,signal));

Internal, not to be called by the application;
OS tick interrupt function (vector name must be defined in FabOS.h)

*11.2. void OS_CreateTaskInt(void (*t)(), uint8_t taskNum, uint8_t *stack, uint8_t stackSize)*

Internal, not to be called by the application;
Create a task, function provided as a macro for simplification of the stack creation and assignment.

11.3. int8_t OS_GetNextTaskNumber()

Internal: determine the next task to be run according priority.

11.4. void OS_Reschedule()__attribute__ ((naked))

Trigger re-scheduling according the task priorities.
Mostly used internally. Could be used externally, but more or less useless, as most OS-API trigger a reschedule anyway and priority of tasks never changes without using OS-API.

12. Helper functions

This functions are available, if the functionality in FabOS_config.h is activated.

12.1. void OS_ErrorHook(uint8_t ErrNo)

For error detection if OS_USEEXTCHECKS == 1 is used, a function of this name has to be provided. An example may look as follows:

```
void OS_ErrorHook(uint8_t ErrNo)
{
	switch(ErrNo)
	{
		case 2:
			// OS_WaitEvent: waiting in idle is not allowed
			break;
		case 4:
			// OS_WaitAlarm: waiting in idle is not allowed
			break;
		case 5:
			// OS_MutexGet: invalid Mutex number
			break;
		case 6:
			// OS_MutexRelease: invalid Mutex number
			break;
		case 7:
			// OS_Alarm misconfiguration
			break;
		case 8:
			// OS_Alarm misconfiguration / ID bigger than array size
			break;
		case 9:
			// OS_WaitAlarm: Alarm is not assigned to the task
			break;
		default:
			break;
	}
}
```

The code may be modified as needed.

12.2. uint16_t OS_GetUnusedStack (uint8_t TaskID)

Get the unused stack space in bytes for any task.
More precise, it gets the stack space, which still contains OS_UNUSEDMASK, so it indicates the minimum free stack.
The parameter is the Task ID. For the idle task specify NUMTASKS.
Do not use this function cyclically, because it consumes a lot of CPU power.

12.3. void OS_GetTicks(uint32_t* pTime)

Get the elapsed OS-ticks (count of elapsed OS-ISR events)
So it can be used to provide a precise time information in e.g. millisecond ticks.

12.4. uint8_t OS_GetQueueSpace(OS_Queue_t pQueue)*

Get the number of unused queue elements in a queue.
The parameter is a pointer to the queue-struct.

```
space = OS_GetQueueSpace(&TestQ);
```

13. Programming hints

13.1. Create a cyclic task

To create a cyclic task, just set an alarm for the next activation directly after waiting for it.
"myOS.CurrTask" could be replaced by the task ID.

```
void myCyclicTask()
{
	OS_CreateAlarm(OSALMCyclicRepeat, OSTSKCyclic); // maybe in main()

	OS_SetAlarm(OSALMCyclicRepeat,10);
	while(1)
	{
		OS_WaitAlarm(OSALMCyclicRepeat);
		OS_SetAlarm(OSALMCyclicRepeat,10);

		// this is a cyclic task, which runs exactly every 10 ticks

		// TODO add your code here
	}
}
```

Attention: do not use the same alarm again within this task.
To have cyclic tasks, which can wait for alarms, use another Alarm bound to this task to provide further delays within this task.

14. Program example

This is the main.c of a onboard datalogger.
The Code, which actually does stuff, is omitted here.
The numerical defines for the Alarm IDs start with ALM_
The numerical defines for the Task IDs start with TSK_
The numerical defines for the Mutexes start with Mutex

```
/*
	GPS-Logger

	Features:
	* RTC Time-Keeping
	* GPS serial analysis NMEA
	* g-meter analog
	* 4 analog inputs
	* DS18S20 temperature sensor
	* Power good sensing
	* SD-card
	* LCD

	Tasks:

	LCD (1s)
	DS18S20 acquisition (bit-banging)
	AD conversion including g-meter (+int)
	SD-card writing with power good from ISR
	NMEA GPS-analysis (+int)
	Idle: RTC
*/

#include "OS/FabOS.h"
#include "nmea.h"
#include "main.h"

OS_DeclareTask(TaskSD,200);
OS_DeclareTask(TaskDS1820,200);
OS_DeclareTask(TaskADC,200);
OS_DeclareTask(TaskLCD,200);
OS_DeclareTask(TaskNMEA,200);

OS_DeclareQueue(NMEAQueue,50,1); // type: OS_Queue_t

#define MutexIO 0
#define MutexGPS 1

// Other stuff

struct
	{
		uint16_t ADC1;
		uint16_t ADC2;
		uint16_t ADC3;
		uint16_t ADC4;
		uint16_t Acc_X;
		uint16_t Acc_Y;
		uint16_t Temp1; // DS18s20
		uint16_t Temp2;
		uint16_t Voltage;
	} MyIOs;

GpsInfo_t MyGPS;
void CPU_init(void);

int main(void)
{
	CPU_init();

	OS_CreateTask(TaskSD,TID_SD);
```

```
        OS_CreateTask(TaskDS1820,TID_DS);
        OS_CreateTask(TaskADC,TID_ADC);
        OS_CreateTask(TaskLCD,TID_LCD);
        OS_CreateTask(TaskNMEA,TID_NMEA);

        OS_CreateAlarm(ALM_ADC, TID_ADC);
        OS_CreateAlarm(ALM_LCD, TID_LCD);
        OS_CreateAlarm(ALM_Timeout, TID_NMEA);

        OS_StartExecution();
        while(1)
        {
                // take care of the RTC
        }
}

void TaskSD(void)
{
        while(1)
        {
                OS_WaitEvent(EVNT_Go); // sent by ADC

                OS_MutexGet(MutexGPS);
                // do write actual coordinates
                OS_MutexRelease(MutexGPS);

                OS_MutexGet(MutexIO);
                // do write the stuff
                OS_MutexRelease(MutexIO);
        }
}

void TaskDS1820(void)
{
        while(1)
        {
                OS_WaitEvent(EVNT_Go); // sent by LCD, 1s beat
                // do DS18s20 acquisition

                //use
                OS_WaitTicks(75);  //750ms conversion time

                OS_MutexGet(MutexIO);
                // set values
                OS_MutexRelease(MutexIO);
        }
}

void TaskADC(void)
{
        OS_SetAlarm(ALM_ADC,10); // every 100ms
        while(1)
        {
                OS_WaitAlarm(ALM_ADC);
                OS_SetAlarm(ALM_ADC,10); // every 100ms

                OS_MutexGet(MutexIO);
                // set values
                OS_MutexRelease(MutexIO);
                if(/*big value change and voltage ok*/1)
                {
                        OS_SetEvent(TID_SD,EVNT_Go);
                }
        }
}

void TaskLCD(void)
{
        OS_SetAlarm(ALM_LCD,100); // every second
        while(1)
        {
                OS_WaitAlarm(ALM_LCD);
                OS_SetAlarm(ALM_LCD,100); // every second
```

```
                OS_MutexGet(MutexIO);
                // display stuff
                OS_MutexRelease(MutexIO);

                OS_MutexGet(MutexGPS);
                // display actual coordinates
                OS_MutexRelease(MutexGPS);

                OS_SetEvent(TID_DS,EVNT_Go); // wake up the DS18s20 Task
        }
}

void TaskNMEA(void)
{
        uint8_t byte,ret,i;
        uint8_t packet[50];

        while(1)
        {
                ret = OS_WaitEventTimeout(EVNT_Go|EVNT_Error, ALM_Timeout, 200);
                // sent by ISR, new message complete, if no Message, timeout.after 2 s
                if(ret == 0 || ret & EVNT_Error)
                {
                        while(!OS_QueueOut(&NMEAQueue, &byte)); // empty the queue

                        // invalidate GPS data
                }
                else
                {
                        //operate queue
                        for(i=0;i<sizeof(packet);i++)
                        {
                                if(!OS_QueueOut(&NMEAQueue, &byte))
                                {
                                        packet[i] = byte;
                                }
                                else
                                {
                                        break;
                                }
                        }
                        //process message byte...
                        OS_MutexGet(MutexGPS);

                        // add actual coordinates
                        nmeaProcessPacket((char*)packet,&MyGPS);

                        OS_MutexRelease(MutexGPS);
                }
        }
}

void CPU_init(void)
{
        // init cyclic ISR
        TCCR1A = 0b00000000;
        TCCR1B = 0b00000010; //1250 kHZ timer ck
        OCR1A  = 12500; //interrupt every 10ms at 10MHz
        TIMSK1 |= 1<<OCIE1A;
}

#if OS_USEEXTCHECKS == 1
void OS_ErrorHook(uint8_t ErrNo)
{
        // you custom error handling
}
#endif
```

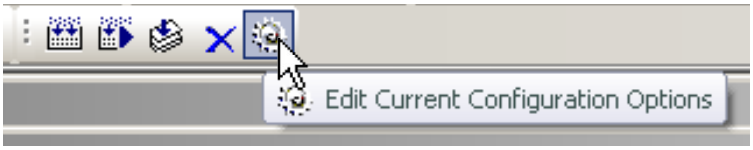
15. Building FabOS with an object file

FabOS is released for evaluation, education and private purposes as an object file.
With a commercial license, you also obtain the full source code.

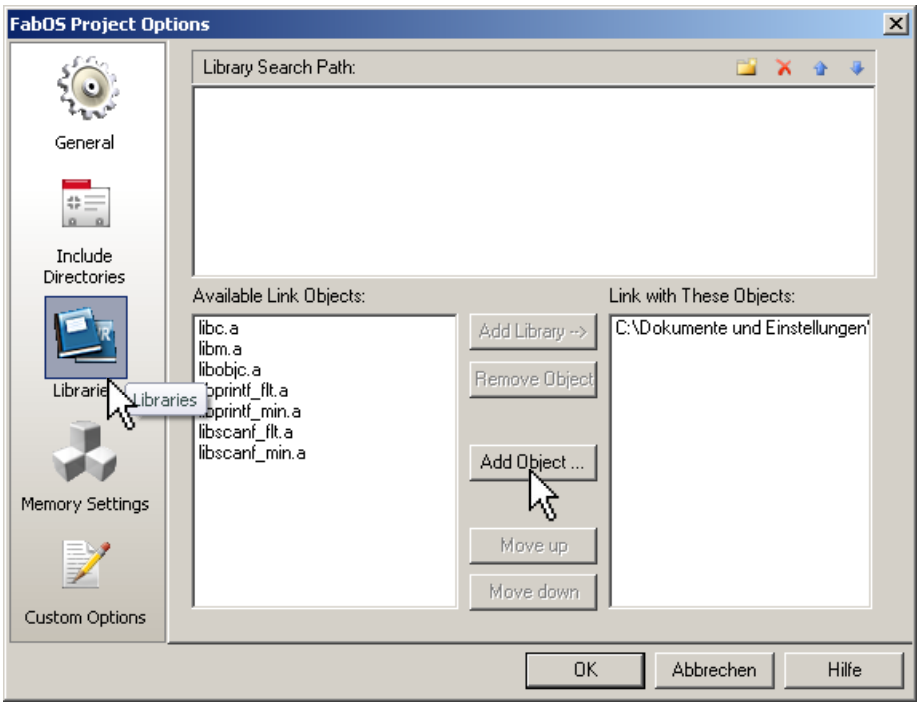
The object file is compiled with the WinAVR GNU compiler version 20100110 with the configuration of the sample FabOS_config.h file.
Especially changing the OS tick timer interrupt source will have no effect.

If it is necessary to have the object file for another compiler or version, please contact the author.

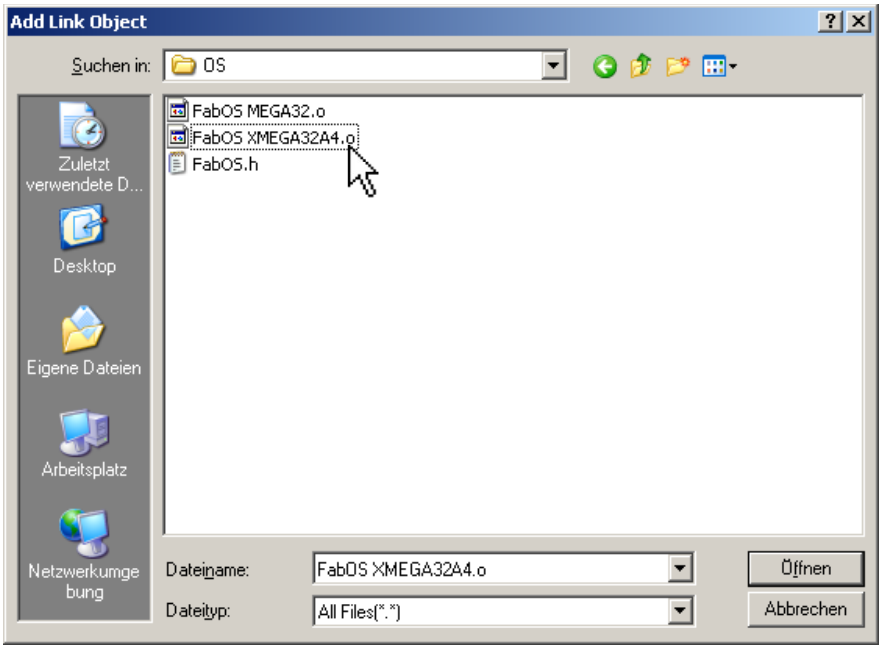
15.1. Step 1: Select Configuration options



15.2. Step 2: Select "Libraries" and "Add Object"



15.3. Step 3: Select “FabOS.o”



Select the FabOS.o file, which best matches your processor.
Contact the author via the contact form at www.huslik-elektronik.de, if you can not find your processor listed.

15.4. Step 4: Build FabOS

If there are no more errors in the project, it should build now.