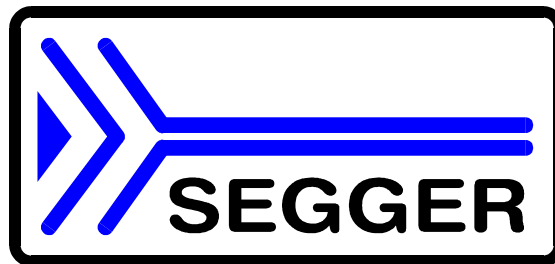


# ***embOS***

Real Time Operating System

CPU & Compiler specifics for  
ARM Cortex M3 core  
using KEIL MDK

Document Rev. 1



A product of Segger Microcontroller Systeme GmbH

**[www.segger.com](http://www.segger.com)**



# Contents

Contents .....	3
1. About this document .....	4
1.1. How to use this manual .....	4
2. Using <b>embOS</b> with Keil MDK .....	5
2.1. Installation .....	5
2.2. First steps .....	6
2.3. The sample application Start_2TaskDisplay.c .....	7
2.4. Stepping through the sample application .....	8
3. Build your own application .....	12
3.1. Required files for an <b>embOS</b> application .....	12
3.2. Change library mode .....	12
3.3. Select an other CPU .....	13
4. CM3 specifics .....	14
4.1. CPU modes .....	14
4.2. Available libraries .....	14
4.3. Task stack for Cortex M3 .....	15
4.4. System stack for Cortex M3 .....	15
4.5. Interrupt stack for Cortex M3 .....	15
5. Interrupts .....	16
5.1. What happens when an interrupt occurs? .....	16
5.2. Defining interrupt handlers in "C" .....	16
5.3. Interrupt vector table .....	16
5.4. Interrupt-stack switching .....	17
5.5. Fast interrupts with Cortex M3 .....	17
5.6. Interrupt priorities .....	17
5.7. Interrupt handling with vectored interrupt controller .....	18
5.7.1. OS_ARM_InstallISRHandler(): Install an interrupt handler .....	18
5.7.2. OS_ARM_EnableISR(): Enable specific interrupt .....	18
5.7.3. OS_ARM_DisableISR(): Disable specific interrupt .....	19
5.7.4. OS_ARM_ISRSetPrio(): Set priority of specific interrupt .....	19
5.8. High priority non maskable exceptions .....	19
6. STOP / WAIT Mode .....	21
7. Technical data .....	21
7.1. Memory requirements .....	21
8. Files shipped with <b>embOS</b> .....	21
9. Index .....	22

# 1. About this document

This guide describes how to use **embOS** Real Time Operating System for the ARM Cortex M3 series of microcontroller using *Keil MDK*.

## 1.1. How to use this manual

This manual describes all CPU and compiler specifics of **embOS** using ARM CM3 based controllers with *Keil MDK*. Before actually using **embOS**, you should read or at least glance through this manual in order to become familiar with the software.

Chapter 2 gives you a step-by-step introduction, how to install and use **embOS** for ARM using *Keil MDK*. If you have no experience using **embOS**, you should follow this introduction, because it is the easiest way to learn how to use **embOS** in your application.

Most of the other chapters in this document are intended to provide you with important detailed information about functionality and fine-tuning of **embOS** for the ARM CM3 based controllers using *Keil MDK*.

## 2. Using **embOS** with Keil MDK

The following chapter describes how to start with and use **embOS** for CM3 and Keil MDK compiler. You should follow these steps to become familiar with **embOS** for ARM and *Keil MDK*

### 2.1. Installation

**embOS** is shipped on CD-ROM or as a zip-file in electronic form.

In order to install it, proceed as follows:

If you received a CD, copy the entire contents to your hard-drive into any folder of your choice. When copying, please keep all files in their respective sub directories. Make sure the files are not read only after copying.

If you received a zip-file, please extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using *Keil MDK* to develop your application, no further installation steps are required. You will find a prepared sample start application, which you should use and modify to write your application. So follow the instructions of the next chapter 'First steps'.

You should do this even if you do not intend to use *Keil MDK* for your application development in order to become familiar with **embOS**.

If for some reason you will not work with the *Keil MDK*, you should:

Copy either all or only the library-file that you need to your work-directory. This has the advantage that when you switch to an updated version of **embOS** later in a project, you do not affect older projects that use **embOS** also.

**embOS** does in no way rely on *Keil MDK*, it may be used without *Keil MDK* using batch files or a make utility without any problem.

## 2.2. First steps

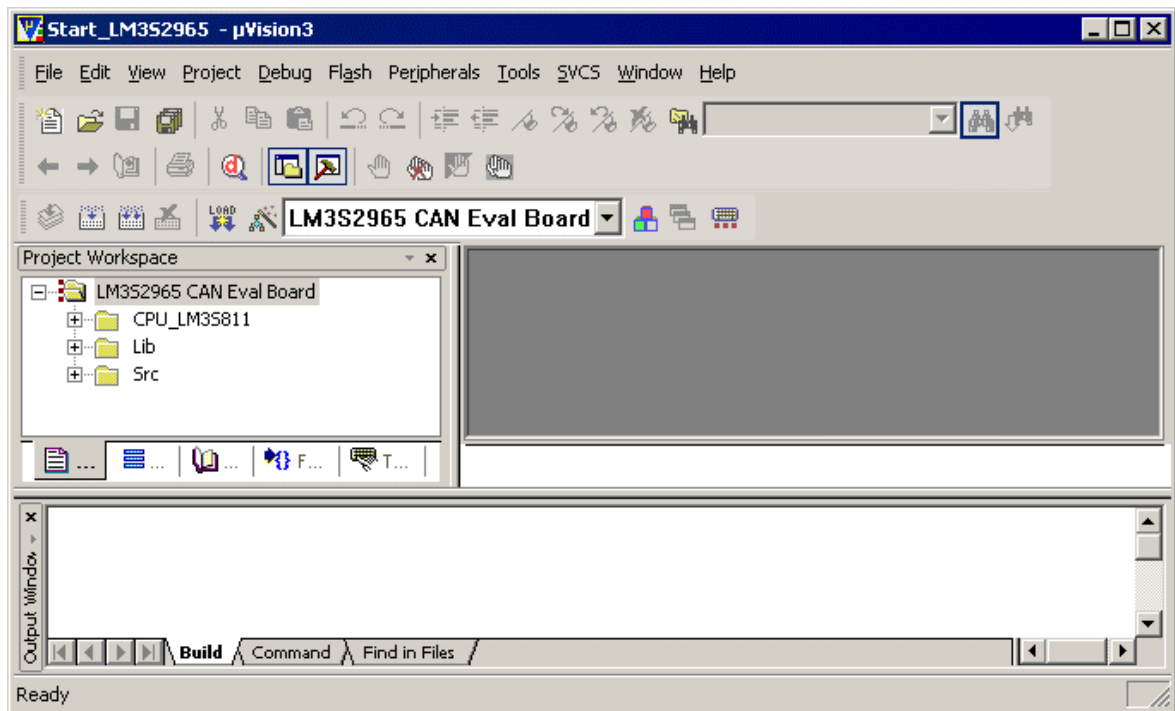
After installation of **embOS** (→ Installation) you are able to create your first multitasking application. You received ready to go sample start workspaces and projects and it is a good idea to use one of these as a starting point of all your applications.

Your **embOS** distribution contains one folder “Start” which contains the sample start workspaces and projects and every additional files used to build your application.

To get your new application running, you should proceed as follows:

- Create a work directory for your application, for example c:\work
- Copy the whole folder ‘Start’ which is part of your **embOS** distribution into your work directory
- Clear the read only attribute of all files in the new ‘start’ folder.
- Open a sample workspace start\start\_\*.uv2 with *Keil MDK* (e.g. by double clicking it). We used Start\_LM3S2965 for our documentation.
- Build the start project

Your screen should look like follows:



For latest information you should refer to the ReadMe.txt files in the start and CPU folders..

## 2.3. The sample application Start\_2TaskDisplay.c

The following is a printout of the sample application Start\_2TaskDisplay.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of **embOS** may look slightly different from this one)

What happens is easy to see:

After initialization of **embOS**, two tasks are created and started

The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```

/*****
*                               SEGGER MICROCONTROLLER SYSTEME GmbH                               *
*                               Solutions for real time microcontroller applications                       *
*****/

-----
File      : Start_2TaskDisplay.c
Purpose  : Sample program for OS running on Luminary Ecx-LM3S2965.
----- END-OF-HEADER -----*/

#include "RTOS.h"
#include "osram128x64x4.h"

OS_STACKPTR int StackHP[256], StackLP[256];          /* Task stacks */
OS_TASK TCBHP, TCBLP;                                /* Task-control-blocks */

void DISPLAY_String(const char * sText, unsigned int vPos, unsigned int hPos);
// Avoid "No Prototype" warning
void DISPLAY_Init(void);                             // Avoid "No Prototype" warning

static OS_RSEMA _SemaDisplay;
static OS_U8    _IsInitd;

static void _InitIfRequired(void) {
    if (_IsInitd == 0) {
        OS_CREATERSEMA(&_SemaDisplay);
        OSRAM128x64x4Init(1000000);
        OSRAM128x64x4StringDraw("embOS on LM3S2965", 16, 20, 11);
        OSRAM128x64x4StringDraw("www.segger.com"    , 26, 30, 11);
    }
    _IsInitd = 1;
}

static void _Lock(void) {
    _InitIfRequired(); // Perform automatic initialisation so that
    explicit call to _Init is not required
    OS_Use(&_SemaDisplay);
}

static void _Unlock(void) {
    _InitIfRequired(); // Perform automatic initialisation so that
    explicit call to _Init is not required
    OS_Unuse(&_SemaDisplay);
}

void DISPLAY_String(const char * sText, unsigned int vPos, unsigned int hPos)
{
    _Lock();
    OSRAM128x64x4StringDraw(sText, vPos, hPos, 11);
    _Unlock();
}

void DISPLAY_Init(void) {
    _InitIfRequired();
}

static void HPTask(void) {
    while (1) {

```

```

    DISPLAY_String("HP Task", 0, 6);
    OS_Delay(50);
    DISPLAY_String("                    ", 0, 6);
    OS_Delay(50);
}
}

static void LPTask(void) {
    while (1) {
        DISPLAY_String("LP Task", 0, 44);
        OS_Delay(200);
        DISPLAY_String("                    ", 0, 44);
        OS_Delay(200);
    }
}

int main(void) {
    OS_IncDI();                /* Initially disable interrupts */
    OS_InitKern();              /* initialize OS */
    OS_InitHW();                /* initialize Hardware for OS */
    DISPLAY_Init();
    /* You need to create at least one task before calling OS_Start() */
    OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
    OS_CREATETASK(&TCBLP, "LP Task", LPTask, 50, StackLP);
    OS_Start();                 /* Start multitasking */
    return 0;
}

```

## 2.4. Stepping through the sample application

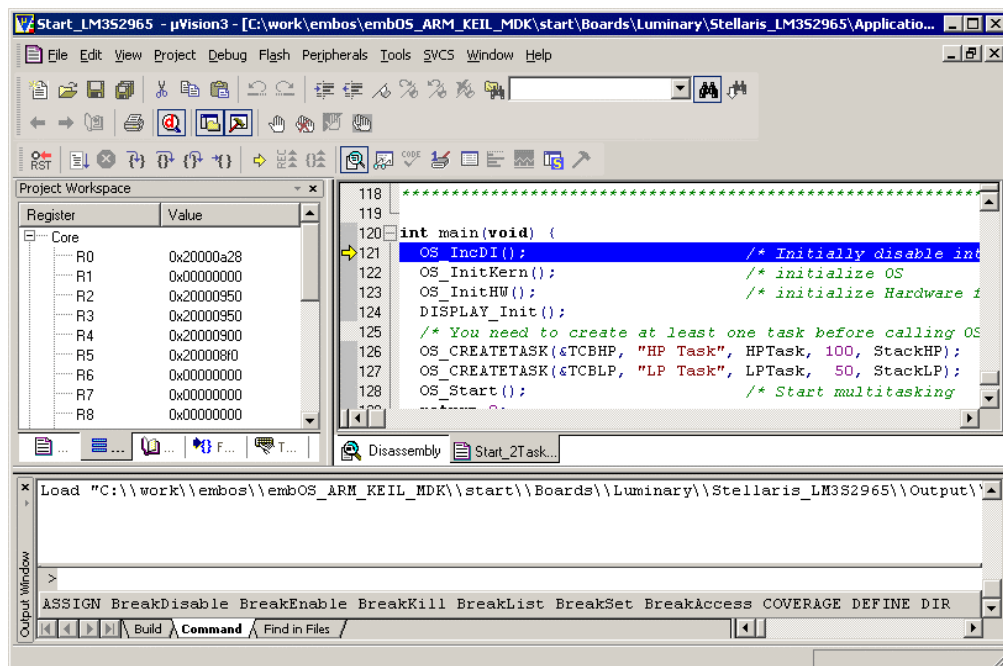
When starting the debugger, you will usually see the main function (very similar to the screenshot below). In some debuggers, you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.

`OS_IncDI()` initially disables interrupts.

`OS_InitKern()` is part of the **embOS** library; you can therefore only step into it in disassembly mode. It initializes the relevant OS-Variables. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.

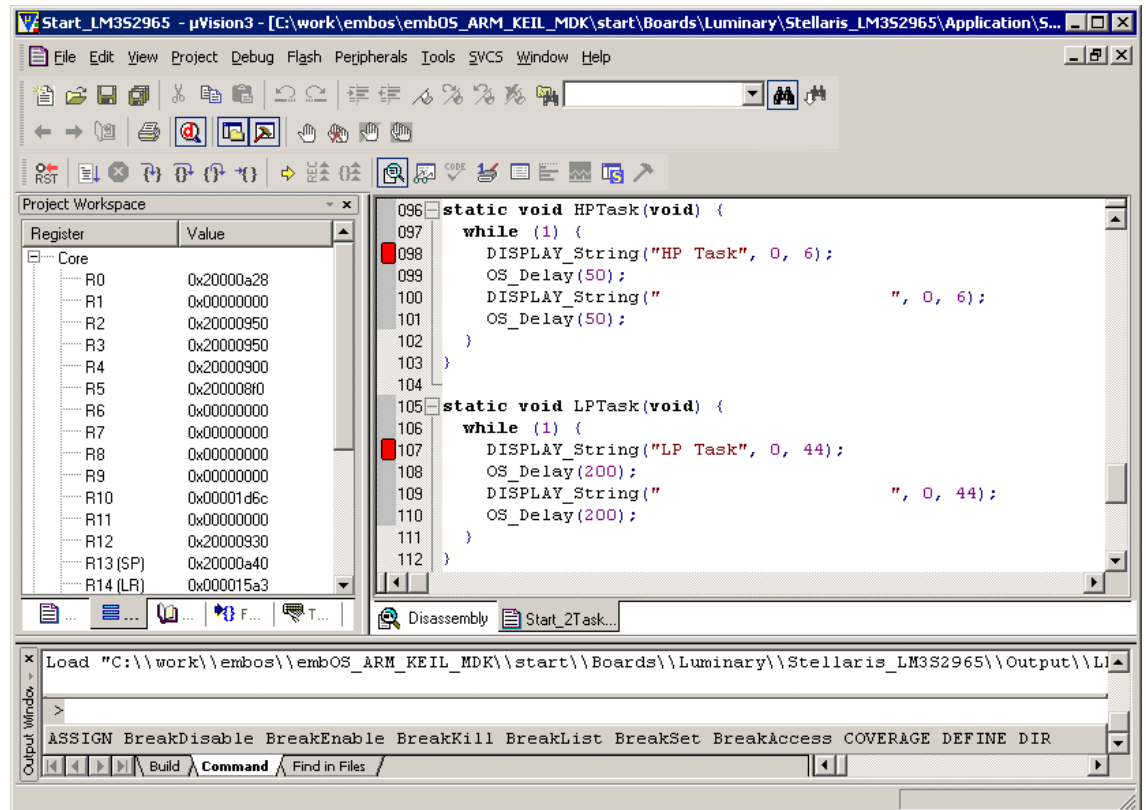
`OS_InitHW()` is part of `RTOSInit_*.c` and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for **embOS**. Step through it to see what is done.

`OS_Start()` should be the last line in main, since it starts multitasking and does not return.

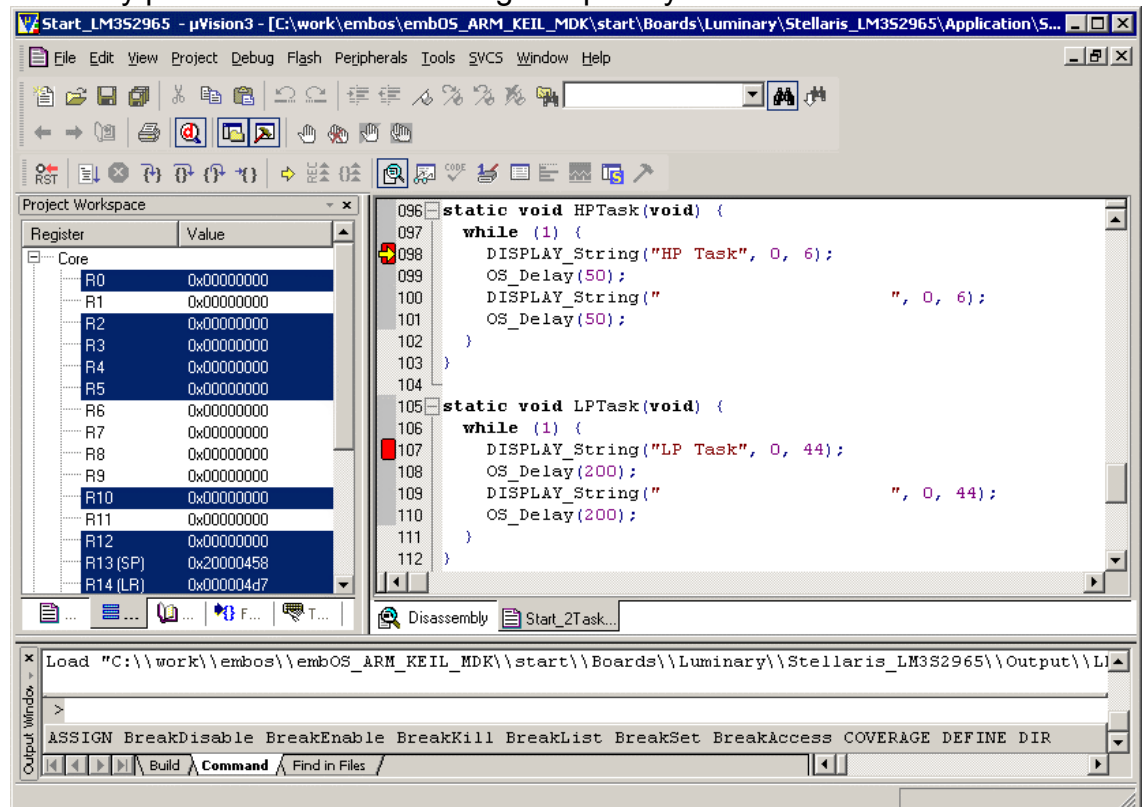




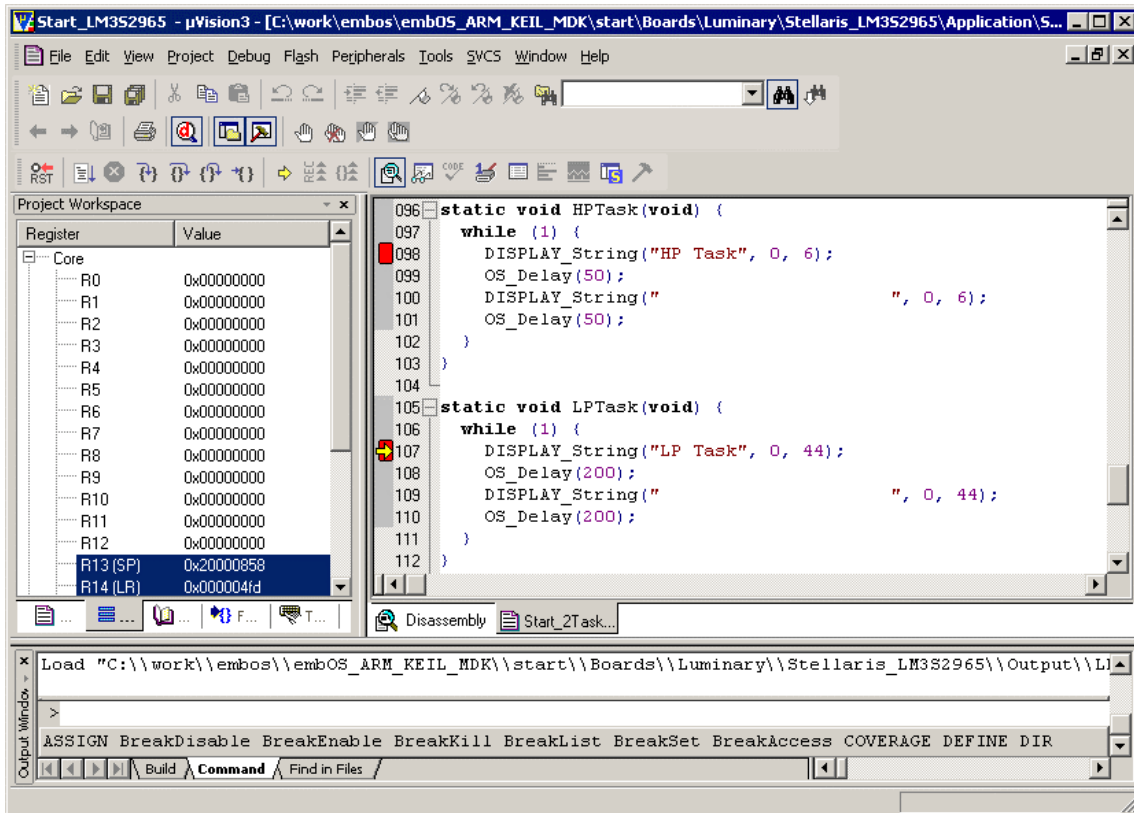
Before you continue stepping, you should set two break points in the two tasks as shown below:



As OS\_Start() is part of the **embOS** library, you can not step through it. You may press GO to reach the highest priority task.

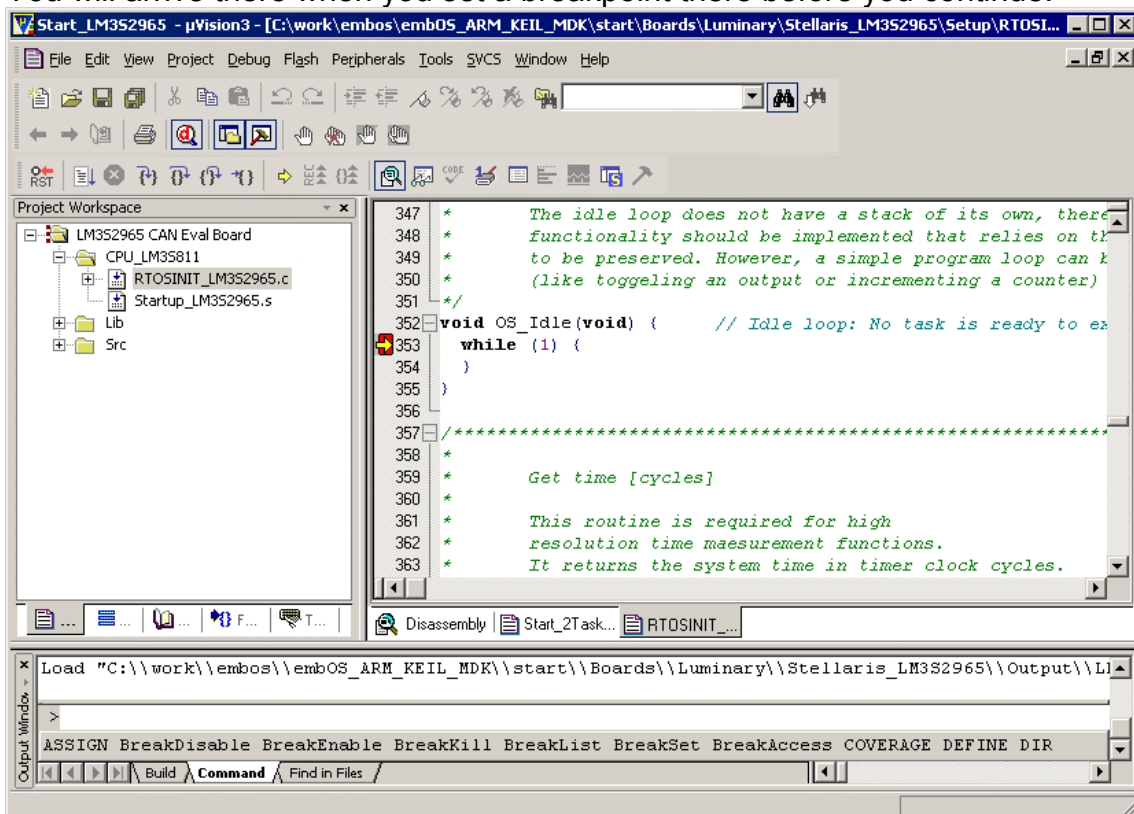


If you continue by pressing GO, you will arrive in the task with lower priority:



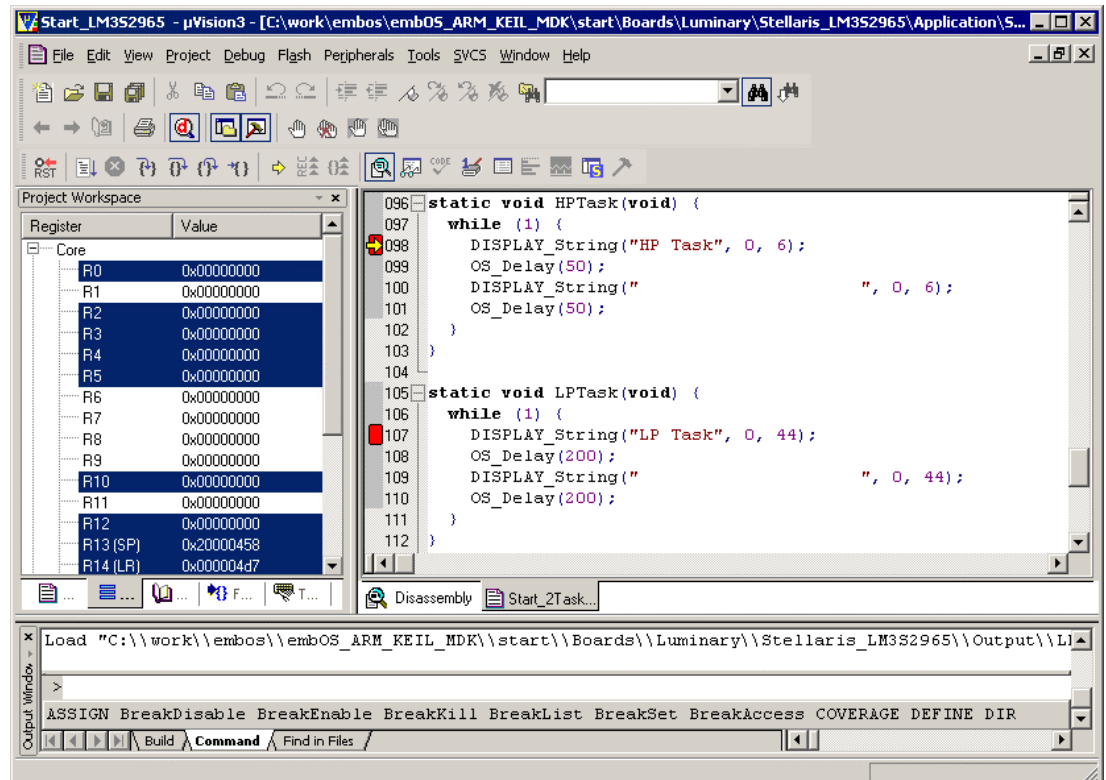
Continuing to step through the program, there is no other task ready for execution. **embOS** will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

You will arrive there when you set a breakpoint there before you continue:



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. Press GO to enter the highest priority task again.

As can be seen by the value of **embOS** timer variable `OS_Time` Task0 continues operation after expiration of the 50 ms delay.



## 3. Build your own application

To build your own application, you should always start with a copy of the sample start workspace and project. Therefore copy the entire folder “Start” from your **embOS** distribution into a working folder of your choice and then modify the start project there. This has the advantage, that all necessary files are included and all settings for the project are already done.

### 3.1. Required files for an **embOS** application

To build an application using **embOS**, the following files from your **embOS** distribution are required and have to be included in your project:

- **RTOS.h** from sub folder Inc\  
This header file declares all **embOS** API functions and data types and has to be included in any source file using **embOS** functions.
- **RTOSInit\_\*.c** from one CPU subfolder.  
It contains hardware dependent initialization code for **embOS** timer and optional UART for embOSView.
- One **embOS library** from the Lib\ subfolder
- **Startup\_\*.s** from the \Setup subfolder.  
It contains the interrupt vector table and default interrupt handler.
- **OS\_Error.c** from subfolder Src\ The error handler is used if any library other than Release build library is used in your project.
- Additional low level init code may be required according to CPU.

When you decide to write your own startup code or use a `__low_level_init` function, please ensure that non initialized variables are initialized with zero, according to “C” standard. This is required for some **embOS** internal variables.

Also ensure, that main is called with CPU running in thread mode using the main stack.

Your main() function has to initialize **embOS** by call of `OS_InitKern()` and `OS_InitHW()` prior any other **embOS** functions except `OS_IncDI()` are called.

You should then modify or replace the main.c source file in the subfolder Src\.

### 3.2. Change library mode

For your application you may wish to choose an other library. For debugging and program development you should use an **embOS**-debug library. For your final application you may wish to use an **embOS**-release library or a stack check library.

Therefore you have to select or replace the **embOS** library in your project or target:

- If your library is already contained in your project, just select the appropriate configuration or enable the library and disable others.
- To add a library, you may add a new embOSLib group to your project and add the new library to the new group. Exclude all other library groups from build, delete unused embOSLib groups or remove them from the configuration. Alternatively you may add the library to the predefined “Lib” group and exclude all other libraries from build.
- Check and set the appropriate `OS_LIBMODE_*` define as preprocessor option.

### 3.3. Select an other CPU

**embOS** for CM3 and Keil MDK compiler contains CPU specific code for various CM3 CPUs and starter kits. The sample start workspaces contain a project for a specific eval boards or starter kits.

Check whether your CPU and starter board is supported by **embOS**. CPU specific functions are located in the board subfolders of the start project folder.

To select a CPU which is already supported, just select the appropriate project from the start workspace.

If your CPU is currently not supported, examine all RTOSInit files in the CPU specific subfolders and select one which almost fits your CPU. You may have to modify OS\_InitHW(), OS\_COM\_Init() and communication routines to embOS-View.

## 4. CM3 specifics

### 4.1. CPU modes

**embOS** supports nearly all memory and code model combinations that KEILs ARM MDK C-Compiler supports.

### 4.2. Available libraries

**embOS** for KEIL ARM MDK is shipped with 24 different libraries, one for each CPU mode / CPU core / endian mode and library type combination.

The libraries are named as follows:

**osT7<c><e><LibMode>.lib**

Parameter	Meaning	Values
<b>e</b>	Endian mode	L: Little
		B: Big
<b>LibMode</b>	Library mode	R: Release
		S: Stack check
		D: Debug
		SP: Stack check + profiling
		DP: Debug + profiling
		DT: Debug + trace

Example:

osT7LR.lib the library for a project using THUMB2 mode, Cortex M3 core, little endian mode and release build library type.

### 4.3. Task stack for Cortex M3

All **embOS** tasks execute in *thread mode* using the process stack pointer. The stack-size required is the sum of the stack-size of all routines plus basic stack size plus size used by exceptions.

The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by **embOS**-routines.

For the Cortex M3 CPU, this minimum task stack size is about 72 bytes.

But because any function call uses some amount of stack and every exception also pushes at least 32 bytes onto the current stack, the task stack size has to be large enough to handle all nested exceptions too. We recommend at least 256 bytes stack as a start.

### 4.4. System stack for Cortex M3

The **embOS** system executes in *thread mode*, the scheduler executes in *handler mode*. The minimum system stack size required by **embOS** is about 136 bytes (stack check & profiling build) However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), and because software-timers and "C"-level interrupt handlers also use the system-stack, the actual stack requirements depend on the application.

### 4.5. Interrupt stack for Cortex M3

If a normal hardware exception occurs, the CM3 core switches to handler mode mode, which uses the main stack pointer. With embOS, the main stack pointer is initialized to use the CSTACK which is defined in the linker command file. A separate IRQ\_STACK is not used, interrupts run on the system stack.

## 5. Interrupts

The Cortex M3 core comes with an built in vectored interrupt controller which supports up to 496 separate interrupt sources. The real number of interrupt sources depends on the specific target CPU.

### 5.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request form the interrupt controller.
- As soon as the interrupts are enabled, the interrupt is executed
- The CPU pushes temporary registers and the return address onto the current stack.
- The CPU switches to handler mode and main stack.
- The CPU saves an exception return code and current flags onto the main stack.
- The CPU jumps to the vector address delivered by the NVIC
- The interrupt handler is processed.
- The interrupt handler ends with a “return from interrupt” by reading the exception return code.
- The CPU switches back to the mode and stack which was active before the exception was called.
- The CPU restores the temporary registers and return address from the stack and continues the interrupted function.

### 5.2. Defining interrupt handlers in "C"

Interrupt handlers for Cortex M3 are written as normal “C”-functions which do not take parameters and do not return any value.

#### Example

"Simple" interrupt-routine

```
void OS_Systick(void) {  
    OS_EnterNestableInterrupt();  
    OS_HandleTick();  
    OS_LeaveNestableInterrupt();  
}
```

### 5.3. Interrupt vector table

After Reset, the ARM Cortex M3 CPU uses an initial interrupt vector table which is located in ROM at address 0x00. It contains the address for the main stack and addresses for all exceptions.

The interrupt vector table is located in the “C” source file Startup\_\*.c in the CPU specific subfolder. All interrupt handler function addresses have to be inserted in the vector table, as long as a RAM vector table is not used.

The vector table may be copied to RAM to enable variable interrupt handler installation. The compile time switch `OS_USE_VARINTTABLE` is used to enable usage of a vector table in RAM. To save RAM, the switch is set to zero per default in `RTOSInit_*.c`. It may be overwritten by project settings to enable the vector table in RAM. The first call of `OS_InstallISRHandler()` will then automatically copy the vector table into RAM.



## 5.4. Interrupt-stack switching

Since CM3 core based controllers have a separate stack pointer for interrupts, there is no need for explicit stack-switching in an interrupt routine. The routines `OS_EnterIntStack()` and `OS_LeaveIntStack()` are supplied for source compatibility to other processors only and have no functionality.

## 5.5. Fast interrupts with Cortex M3

Instead of disabling interrupts when **embOS** does atomic operations, the interrupt level of the CPU is set to 128. Therefore all interrupt priorities higher than 128 can still be processed. Please note, that lower priority number define a higher priority. All interrupts with priority level from 0 to 128 are never disabled. These interrupts are named *Fast interrupts*. You must not execute any **embOS** function from within a *fast interrupt* function.

## 5.6. Interrupt priorities

With introduction of *Fast interrupts*, interrupt priorities useable for interrupts using **embOS** API functions are limited.

- Any interrupt handler using **embOS** API functions has to run with interrupt priorities from 128 to 255.  
These **embOS** interrupt handlers have to start with `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` and must end with `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()`.
- Any *Fast interrupt* (running at priorities from 0 to 127) must not call any **embOS** API function. Even `OS_EnterInterrupt()` and `OS_LeaveInterrupt()` must not be called.
- Interrupt handler running at low priorities (from 128 to 255) not calling any **embOS** API function are allowed, but must not re-enable interrupts!

**The priority limit between **embOS** interrupts and Fast interrupts is fixed to 128 and can only be changed by recompiling **embOS** libraries!**

## 5.7. Interrupt handling with vectored interrupt controller

For Cortex M3, which has a built in vectored interrupt controller, **embOS** delivers additional functions to install and setup interrupt handler functions.

To handle interrupts with the vectored interrupt controller, **embOS** offers the following functions:

### 5.7.1. OS\_ARM\_InstallISRHandler(): Install an interrupt handler

#### Description

OS\_ARM\_InstallISRHandler() is used to install a specific interrupt vector when ARM CPUs with vectored interrupt controller are used.

#### Prototype

```
OS_ISR_HANDLER* OS_ARM_InstallISRHandler (int ISRIndex,
                                           OS_ISR_HANDLER* pISRHandler);
```

Parameter	Meaning
ISRIndex	Index of the interrupt source, normally the interrupt vector number.
pISRHandler	Address of the interrupt handler function.

#### Return value

OS\_ISR\_HANDLER\*: the address of the previous installed interrupt function, which was installed at the addressed vector number before.

#### Add. information

This function just installs the interrupt vector but does not modify the priority and does not automatically enable the interrupt.

When the interrupt vector table should be located in RAM, the first call of this function copies the vector table into RAM and programs the interrupt controller to use the RAM table.

When the interrupt vector table should reside in ROM, the function does nothing and always returns "NULL".

### 5.7.2. OS\_ARM\_EnableISR(): Enable specific interrupt

#### Description

OS\_ARM\_EnableISR() is used to enable interrupt acceptance of a specific interrupt source in a vectored interrupt controller.

#### Prototype

```
void OS_ARM_EnableISR(int ISRIndex)
```

Parameter	Meaning
ISRIndex	Index of the interrupt source which should be enabled.

#### Return value

NONE.

#### Add. information

This function just enables the interrupt inside the interrupt controller. It does not enable the interrupt of any peripherals. This has to be done elsewhere.

### 5.7.3. OS\_ARM\_DisableISR(): Disable specific interrupt

#### Description

OS\_ARM\_DisableISR() is used to disable interrupt acceptance of a specific interrupt source in a vectored interrupt controller which is not of the VIC type.

#### Prototype

```
void OS_ARM_DisableISR(int ISRIndex);
```

Parameter	Meaning
ISRIndex	Index of the interrupt source which should be disabled.

#### Return value

NONE.

#### Add. information

This function just disables the interrupt in the interrupt controller. It does not disable the interrupt of any peripherals. This has to be done elsewhere.

### 5.7.4. OS\_ARM\_ISRSetPrio(): Set priority of specific interrupt

#### Description

OS\_ARM\_ISRSetPrio () is used to set or modify the priority of a specific interrupt source by programming the interrupt controller.

#### Prototype

```
int OS_ARM_ISRSetPrio(int ISRIndex, int Prio);
```

Parameter	Meaning
ISRIndex	Index of the interrupt source which should be modified.
Prio	The priority which should be set for the specific interrupt.

#### Return value

Previous priority which was assigned before the call of OS\_ARM\_ISRSetPrio().

#### Add. information

This function sets the priority of an interrupt channel by programming the interrupt controller. Please refer to CPU specific manuals about allowed priority levels.

## 5.8. High priority non maskable exceptions

High priority non maskable exceptions with non configurable priority like Reset, NMI and HardFault can not be used with **embOS** functions.

These exceptions are never disabled by **embOS**.

Never call any **embOS** function from an exception handler of one of these exceptions.



## 6. STOP / WAIT Mode

In case your controller supports some kind of power saving mode, it should be possible to use it also with **embOS**, as long as the system timer keeps working and timer interrupts are processed. To enter that mode, you usually have to implement some special sequence in function `OS_Idle()`, which you can find in **embOS** module `RTOSINIT.c`.

## 7. Technical data

### 7.1. Memory requirements

These values are neither precise nor guaranteed but they give you a good idea of the memory-requirements. They vary depending on the current version of **embOS**. The kernel itself has a minimum ROM size requirement of about 1.700 bytes.

In the table below, you can find minimum RAM size for **embOS** resources. Please note, that sizes depend on selected **embOS** library mode; table below is for a release build.

<b>embOS</b> resource	RAM [bytes]
Task control block	32
Resource semaphore	8
Counting semaphore	4
Mailbox	20
Software timer	20

## 8. Files shipped with **embOS**

Directory	File	Explanation
root	*.pdf	Generic API and target specific documentation
root	Release.html	Version control document
root	embOSView.exe	Utility for runtime analysis, described in generic documentation
START\INC	RTOS.H	Include file for <b>embOS</b> , to be included in every "C"-file using <b>embOS</b> -functions
START\LIB	os*.lib	<b>embOS</b> libraries
START\BOARDS	*.*	Sample workspace and project files for Keil MDK.

Any additional files shipped serve as example.

## 9. Index

### F

Fast interrupt ..... 17

### H

Halt-mode ..... 21

### I

Idle-task-mode ..... 21

Installation ..... 5

Interrupt priority ..... 17

Interrupt stack ..... 15

Interrupt stack switching ..... 17

Interrupt vector table ..... 16

Interrupt, fast ..... 17

Interrupts ..... 16

Interrupts, FIQ ..... 19

### M

memory models ..... 14

memory requirements ..... 21

### O

OS\_ARM\_DisableISR() ..... 19

OS\_ARM\_EnableISR() ..... 18

OS\_ARM\_InstallISRHandler() ..... 18

OS\_ARM\_ISRSetPrio() ..... 19

OS\_IRQ\_SERVICE() ..... 16

OS\_USE\_VARINTTABLE ..... 16

### S

Stacks, interrupt stack ..... 15

Stacks, system stack ..... 15

Stacks, task stack ..... 15

Stop-mode ..... 21

System stack ..... 15

### T

target hardware ..... 21

Task stack ..... 15

### W

Wait-mode ..... 21