embOS

Real Time Operating System

CPU & Compiler specifics for Cortex M using IAR Embedded Workbench®

Software version 3.86d
Document: UM01014
Revision: 0

Date: May 10, 2012

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER MICROCONTROLLER GmbH & Co. KG (the manufacturer) assumes no responsibility for any errors or omissions. The manufacturer makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. The manufacturer specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller Systeme GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel. +49 2103-2878-0
Fax.+49 2103-2878-28
Email: support@segger.com
Internet: *http://www.segger.com*

Software and manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: May 10, 2012

Software	Manual	Date	By	Description
3.86d	0	120510	AW	Software: New embOS sources V3.86d OS_ExtendTaskContext_TLS_VFP() corrected.

3.84c1	1	120203	AW	Software: Scheduler for CortexM4 with VFP corrected.
				Chapter 4: New functions to save and restore VFP context of Cortex M4 in ISR handler:
				OS_VFP_Save()
				OS_VFP_Restore()

3.84c	0	120117	TS	Software version updated.
3.84.1	0	111103	AW	Software: New scheduler uses main stack for OS_Idle(). Chapter 5: Stacks, system stack description updated. New Chapter 6.8.4 and 6.8.5: Interrupt peripheral identifier and priority values used with embOS described more in detail Chapter 7: Interrupt controller setup using CMSIS described more in detail. New chapter 7.7.1 describes Differences between embOS functions and CMSIS functions

3.84	0	111027	AW	Chapter 3.2.1: New libraries with VFPv4 support added.
				Chapter 4.3, 4.4: Support for VFPv4 added,
				OS_ExtendTaskContext_TLS()
				OS_ExtendTaskContext_TLS_VFP()
				OS_ExtendTaskContext_VFP()

3.82u	0	110701	AW	Chapter CMSIS with IAR EWARM V6 added.
3.82s	0	110323	TS	New library mode DPL added.
3.82m	0	101117	AW	Thread local storage for new IAR workbench V6
3.82l	0	101027	AW	Library names updated for new IAR workbench V6 Thread safe library support modified for IAR workbench V6
3.82h	0	100722	TS	embOS CM3 and embOs CM0 manual merged
3.82a	1	100701	AW	Chapter Stacks: Task stack size corrected.
3.82a	0	091026	TS	First version

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler, workbench)
- The C programming language
- The target processor

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

The intention of this manual is to give you a CPU- and compiler-specific introduction to embOS.

For a quick and easy startup with embOS, refer to Chapter 1 & 2 which include a step-by-step introduction to using embOS.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections

Table 2.1: Typographic conventions

SEGGER Microcontroller GmbH & GmbH develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development-time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficent real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER developes and produces programming tools for flash microcontrollers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin

Graphics software and GUI

emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display. Starterkits, eval- and trial-versions are available.

embOS

Real Time Operating System

embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources. The profiling PC tool embOSView is included.

emFile

File system

emFile is an embedded file system with FAT12, FAT16 and FAT32 support. emFile has been optimized for minimum memory consumption in RAM and ROM while maintaining high speed. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack

USB device stack

A USB stack designed to work on any embedded system with a USB client controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher

Flash programmer

Flash Programming tool primarily for microcontrollers.

J-Link

JTAG emulator for ARM cores

USB driven JTAG interface for ARM cores.

J-Trace

JTAG emulator with trace

USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software

Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Using embOS for Cortex M

This chapter describes how to start with and use embOS for Cortex M cores and the IAR compiler. You should follow these steps to become familiar with embOS.

1.1 Installation

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

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, keep all files in their respective sub directories. Make sure the files are not read only after copying. If you received a zip-file, extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using the IAR Embedded Workbench project manager to develop your application, no further installation steps are required. You will find a lot of prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section *First steps* on page 13.

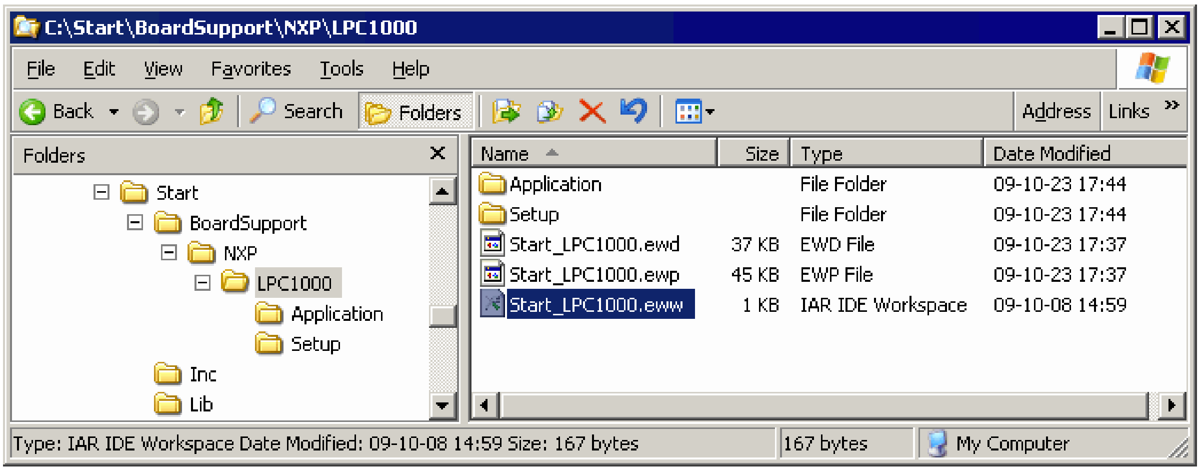
You should do this even if you do not intend to use the project manager for your application development to become familiar with embOS.

If you will not work with the embedded workbench, 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 the IAR Embedded Workbench project manager, it may be used without the project manager using batch files or a make utility without any problem.

1.2 First steps

After installation of embOS you can create your first multitasking application. You received several ready to go sample start workspaces and projects and every other files needed in the subfolder **Start**. It is a good idea to use one of them as a starting point for all of your applications. The subfolder **BoardSupport** contains the workspaces and projects which are located in manufacturer- and CPU-specific subfolders.

For the first step, you may use the project for NXP LPC1113 CPU:



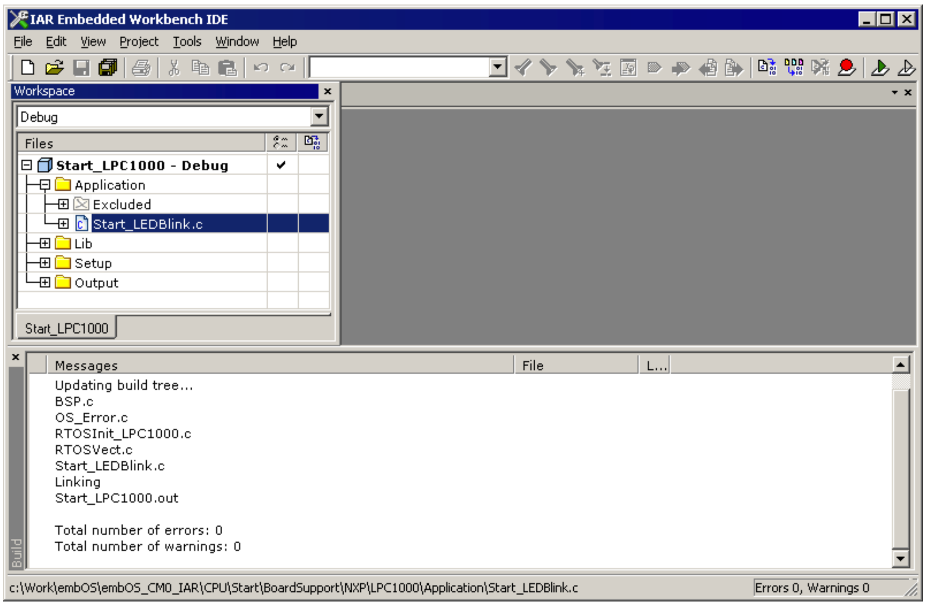
Name	Size	Type	Date Modified
Application		File Folder	09-10-23 17:44
Setup		File Folder	09-10-23 17:44
Start_LPC1000.ewd	37 KB	EWD File	09-10-23 17:37
Start_LPC1000.ewp	45 KB	EWP File	09-10-23 17:37
Start_LPC1000.eww	1 KB	IAR IDE Workspace	09-10-08 14:59

Type: IAR IDE Workspace Date Modified: 09-10-08 14:59 Size: 167 bytes

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 the sample workspace. **Start\BoardSupport\NXP\Start_LPC1000.eww** with the IAR Embedded Workbench project manager (for example, by double clicking it).
- Build the project. It should be build without any error or warning messages.

After generating the project of your choice, the screen should look like this:



```
Updating build tree...
BSP.c
OS_Error.c
RTOSInit_LPC1000.c
RTOSVect.c
Start_LEDBlink.c
Linking
Start_LPC1000.out

Total number of errors: 0
Total number of warnings: 0
```

c:\Work\embOS\embOS_CM0_IAR\CPU\Start\BoardSupport\NXP\LPC1000\Application\Start_LEDBlink.c   Errors 0, Warnings 0

For additional information you should open the `ReadMe.txt` file which is part of every specific project. The ReadMe file describes the different configurations of the project and gives additional information about specific hardware settings of the supported eval boards, if required.

1.3 The example application Start_2Tasks.c

The following is a printout of the example application `Start_2Tasks.c`. It is a good starting point for your application. (Note that the file actually shipped with your port of embOS may look slightly different from this one.)

What happens is easy to see:

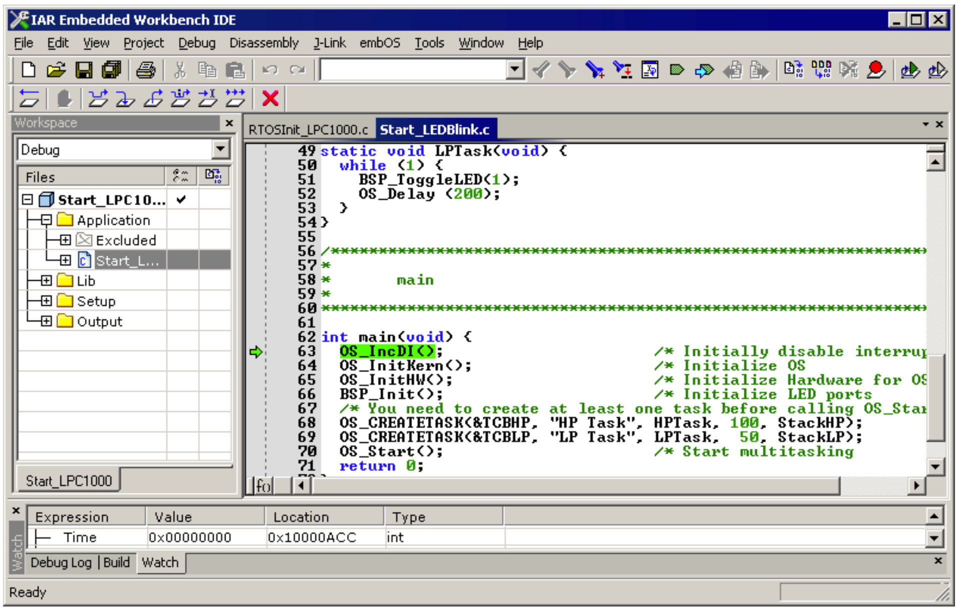
After initialization of embOS; two tasks are created and started.
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/**********************************************************
* SEGGER MICROCONTROLLER SYSTEME GmbH
* Solutions for real time microcontroller applications
***********************************************************
File : Main.c
Purpose : Skeleton program for embOS
--------- END-OF-HEADER ---------------------------------*/

#include "RTOS.H"

OS_STACKPTR int Stack0[128], Stack1[128]; /* Task stacks */
OS_TASK TCB0, TCB1;                       /* Task-control-blocks */'

void HPTask(void) {
  while (1) {
    OS_Delay (10);
  }
}

void LPTask(void) {
  while (1) {
    OS_Delay (50);
  }
}

/**********************************************************
*
* main
*
**********************************************************/

void main(void) {
  OS_IncDI();                            /* Initially disable interrupts */
  OS_InitKern();                         /* Initialize OS */
  OS_InitHW();                           /* Initialize Hardware for OS */
  /* You need to create at least one task here ! */
  OS_CREATETASK(&TCB0, "HP Task", HPTask, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", LPTask, 50, Stack1);
  OS_Start();                            /* Start multitasking */
}
```

1.4 Stepping through the sample application

When starting the debugger, you will see the `main` function (see example screenshot below). The `main` function appears as long as the C-SPY option **Run to main** is selected, which it is by default. Now you can step through the program. `OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the embOS library and written in assembler; 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.

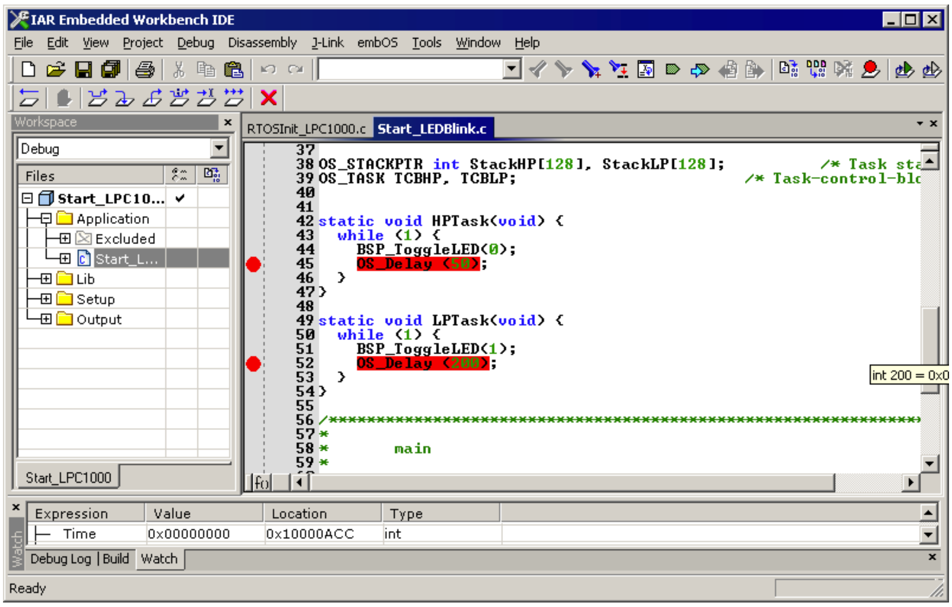
`OS_Start()` should be the last line in main, because it starts multitasking and does not return.



```
static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

/*********************************************************************
*
*       main
*
**********************************************************************

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

Expression	Value	Location	Type
Time	0x00000000	0x10000ACC	int

Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

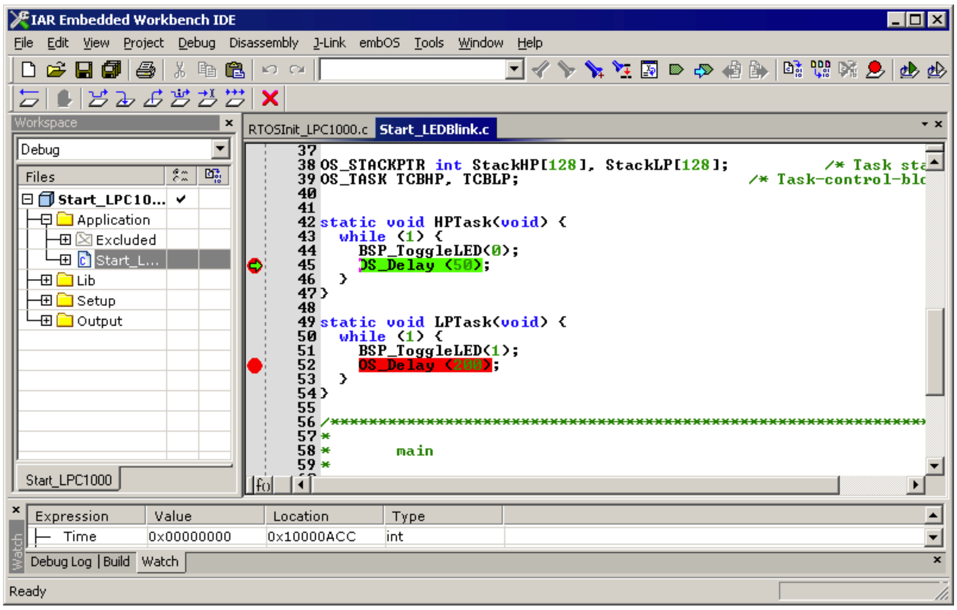


```

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task sta
OS_TASK TCBHP, TCBLP;                             /* Task-control-blc


static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

/*********************************************************************
*
*       main
*
```

Expression	Value	Location	Type
Time	0x00000000	0x10000ACC	int

As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

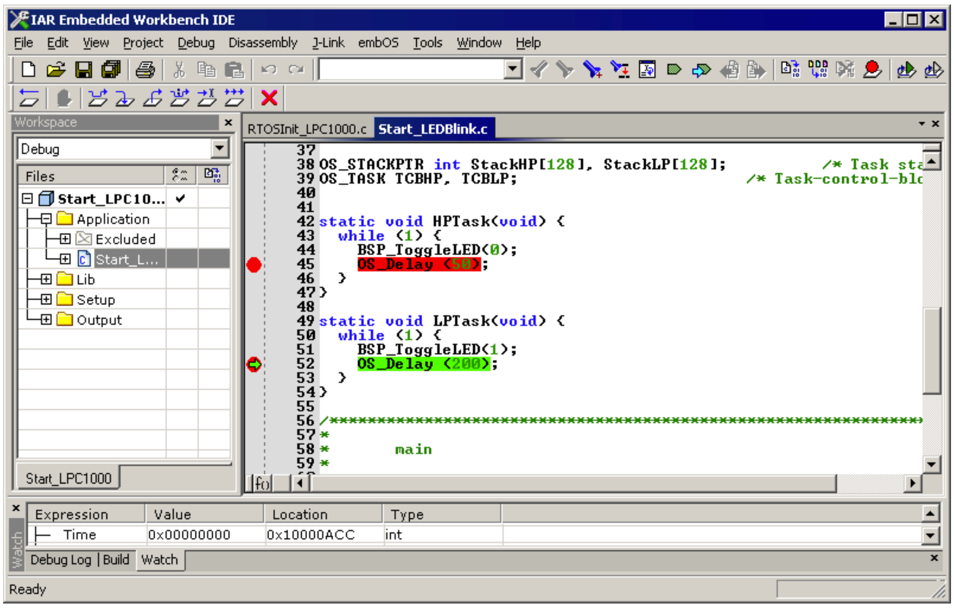
Click **GO**, step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task.



```
OS_STACKPTR int StackHP[128], StackLP[128];          /* Task sta
OS_TASK TCBHP, TCBLP;                              /* Task-control-blo


static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

/*********************************************************************
*
*       main
*
```

Expression	Value	Location	Type
Time	0x00000000	0x10000ACC	int

If you continue stepping, you will arrive in the task that has lower priority:

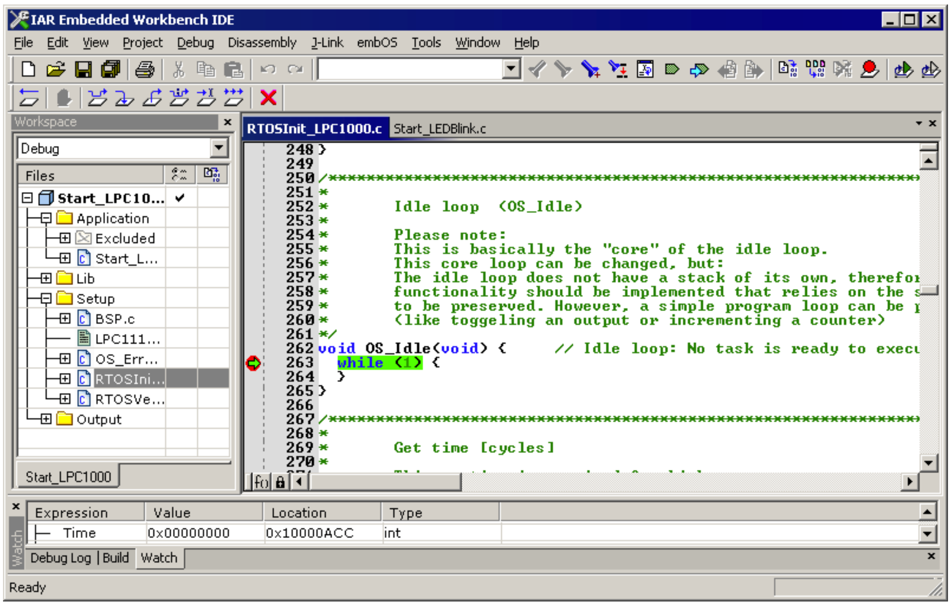


```
OS_STACKPTR int StackHP[128], StackLP[128];          /* Task sta
OS_TASK TCBHP, TCBLP;                              /* Task-control-blo


static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

/*********************************************************************
*
*       main
*
```

Expression	Value	Location	Type
Time	0x00000000	0x10000ACC	int

Continue 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 step into the `OS_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit*.c`. You may also set a breakpoint there before you step over the delay in `LPTask`.

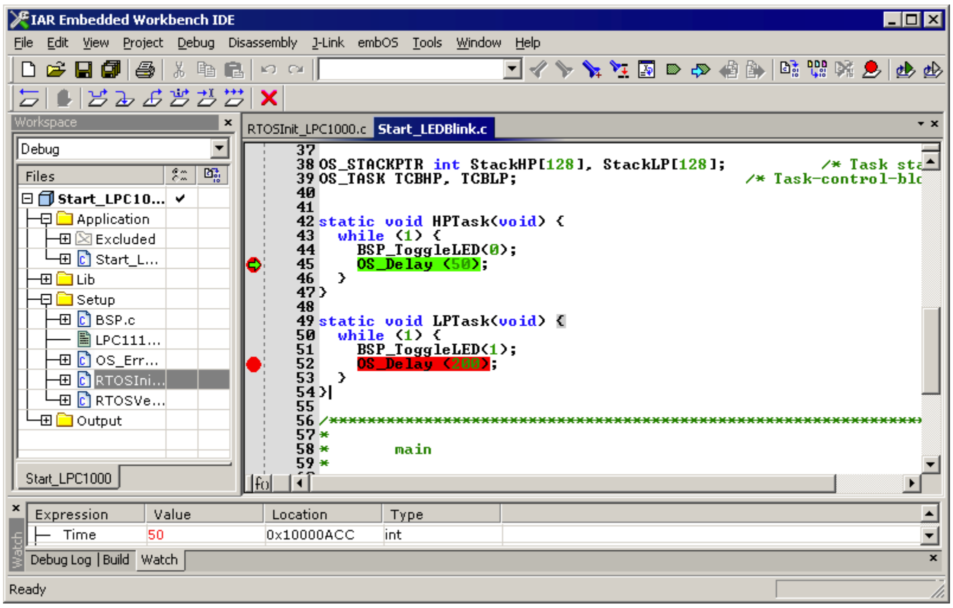


```
}

/*********************************************************************
*
*       Idle loop  (OS_Idle)
*
*       Please note:
*       This is basically the "core" of the idle loop.
*       This core loop can be changed, but:
*       The idle loop does not have a stack of its own, therefor
*       functionality should be implemented that relies on the s
*       to be preserved. However, a simple program loop can be p
*       (like toggeling an output or incrementing a counter)
*/
void OS_Idle(void) {     // Idle loop: No task is ready to execu
  while (1) {
  }
}

/*********************************************************************
*
*       Get time [cycles]
*
```

Expression	Value	Location	Type
Time	0x00000000	0x10000ACC	int

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.

As can be seen by the value of embOS timer variable `OS_Time`, shown in the **Watch** window, `HPTask` continues operation after expiration of the 50 ms delay.



```

OS_STACKPTR int StackHP[128], StackLP[128];          /* Task sta
OS_TASK TCBHP, TCBLP;                               /* Task-control-blc


static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

/*********************************************************************
*
*       main
*
```

Expression	Value	Location	Type
Time	50	0x10000ACC	int

Chapter 2

Build your own application

This chapter provides all information to setup your own embOS project.

2.1 Introduction

To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in First steps on page 9 and modify the project to fit your needs. Using a sample project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS for Cortex M

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 subfolder 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 target specific **BoardSupport\<Manufacturer>\<MCU>** subfolder.
  It contains hardware-dependent initialization code for embOS. It initializes the system timer, timer interrupt and optional communication for embOSView via UART or JTAG.
- One embOS library from the subfolder Lib\.
- OS_Error.c from one target specific subfolder **BoardSupport\<Manufacturer>\<MCU>**.
  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, 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 the CPU running in supervisor or system mode.
Your main() function has to initialize embOS by a call of OS_InitKern() and OS_InitHW() prior any other embOS embOS functions are called.
You should then modify or replace the Start_2Task.c source file in the subfolder Application\.

2.3 Change library mode

For your application you might want to choose another 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 selected library is already available in your project, just select the appropriate configuration.
- To add a library, you may add a new `Lib` group to your project and add this library to the new group. Exclude all other library groups from build, delete unused `Lib` groups or remove them from the configuration.
- Check and set the appropriate `OS_LIBMODE_*` define as preprocessor option and/or modify the OS_Config.h file accordingly.

2.4 Select another CPU

embOS contains CPU-specific code for various Cortex M CPUs. Manufacturer- and CPU specific sample start workspaces and projects are located in the subfolders of the **BoardSupport** folder. To select a CPU which is already supported, just select the appropriate workspace from a CPU specific folder.

If your Cortex M 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 the interrupt service routines for embOS timer tick and communication to embOSView and `__low_level_init()`.

The easiest way to get embOS running on an unsupported CPU is using the generic CMSIS start project and adding the device specific files from the CPU vendor.

Chapter 3

Cortex M version specifics

3.1 CPU modes

embOS for Cortex M supports all memory and code model combinations that IAR C/C++ Compiler supports.

3.2 Available libraries

embOS for Cortex M and IAR compiler comes with 48 different libraries, one for each CPU mode / CPU core / endianess / library mode combination. The library names follow the naming convention for the system libraries from IAR.

3.2.1 Naming conventions for prebuilt libraries compatible to IAR EWARM V6.x

embOS Cortex M for the IAR Embedded Workbench V6 is shipped with different prebuilt libraries with different combinations of the following features:

- Instruction set architecture - `Arch`
- CPU mode - `CpuMode`
- Byte order - `ByteOrder`
- Library mode - `LibMode`

The libraries are named as follows:

```
os<Arch>_<CpuMode><ByteOrder><VFP_support>_<Libmode>.a
```

Parameter	Meaning	Values
`Arch`	Specifies the CPU variant	6m: Cortex M0 7m: Cortex M3 / M4 / M4F
`CpuMode`	Specifies the CPU mode.	t: Always thumb
`ByteOrder`		b: Big endian l: Little endian
`VFP_support`		_: No hardware VFP support v: VFPv4 support (Cortex M4F)
`LibMode`	Specifies the library mode	xr: Extreme Release
		r: Release
		s: Stack check
		sp: Stack check + profiling
		d: Debug
		dp: Debug + profiling
		dpl: Debug + profiling + low optimization
		dt: Debug + profiling + trace

Table 3.1: Naming conventions for prebuild libraries compatible to IAR EW V6.x

Example

os6m_tl__dp.a is the library for a project using a CM0 core, thumb mode, little endian mode with debug and profiling support.

os7m_tlv_dp.a is the library for a project using a CM4F core, thumb mode, little endian mode and VFPv4 floating point unit with debug and profiling support.

Note:

The libraries for Cortex M3 can also be used for Cortex M4 and Cortex M4F targets as long as the VFP is not used in Cortex M4F.
When VFPv4 is selected in the project options, the IAR startup code automatically adds code to enable the VFP after reset.
When using your own startup code, please ensure, the VPF is enabled or diasabled according the project settings.
Using the VFP requires the libraries with VFP support os7m_tlv_xx.

3.2.2 Naming conventions for prebuilt libraries compatible to IAR EWARM V5.x

embOS Cortex M for the IAR Embedded Workbench V5 is shipped with different prebuilt libraries with different combinations of the following features:

- Instruction set architecture - `Arch`
- CPU mode - `CpuMode`
- Byte order - `ByteOrder`
- FPU (all prebuilt libraries are built with software floating-point support.) - `float`
- Interworking - `interwork`
- Stack alignment - `StackAlign`
- Library mode - `LibMode`

The libraries are named as follows:

`os<Arch><CpuMode><ByteOrder><float><interwork><StackAlign><Libmode>.a`

Parameter	Meaning	Values
`Arch`	Specifies the CPU variant	6m: Cortex M0 7m: Cortex M3
`CpuMode`	Specifies the CPU mode.	t: Always thumb
`ByteOrder`		b: Big endian l: Little endian
`float`	Specifies the FPU support	n: Software floating point support
`interwork`	Specifies the interworking	n: Always NO interworking
`StackAlign`	Alignment of stacks	8: Always 8 bytes aligned
`LibMode`	Specifies the library mode	xr: Extreme Release
		r: Release
		s: Stack check
		sp: Stack check + profiling
		d: Debug
		dp: Debug + profiling
		dpl: Debug + profiling + low optimization
		dt: Debug + profiling + trace

Table 3.2: Naming conventions for prebuild libraries compatible to IAR EW V6.x

Example

os6mtlnn8dp.a is the library for a project using a CM0 core, thumb mode, little endian mode, no FPU, no interworking, 8 byte stack alignment with debug and profiling support.

Note:

The libraries for Cortex M3 can also be used for Cortex M4 targets. Cortex M4F with VFPv4 is not supported. embOS libraries for IAR EWARM V5 can not be used with the VFP switched on.

3.2.3 Migrating from EWARM v5 to EWARM v6

The embOS library names have changed in the embOS version for EWARM v6. The embOS library name for the new version can be translated as follows:

1. Add an underscore after the first 4 characters
os7mtlnn8xr -> os7m_tlnn8xr

2. Replace "nn8" by two undrescores
os7m_tlnn8xr -> os7m_tl__xr

Complete list of library names for embOS IAR V5 / V6:

EWARM v5 library name	EWARM v6 library name
Cortex M3, little endian	
os7mtlnn8xr	os7m_tl__xr
os7mtlnn8r	os7m_tl__r
os7mtlnn8s	os7m_tl__s
os7mtlnn8sp	os7m_tl__sp
os7mtlnn8d	os7m_tl__d
os7mtlnn8dp	os7m_tl__dp
os7mtlnn8dt	os7m_tl__dt
os7mtlnn8dpl	os7m_tl__dpl
Cortex M3, big endian	
os7mtbnn8xr	os7m_tb__xr
os7mtbnn8r	os7m_tb__r
os7mtbnn8s	os7m_tb__s
os7mtbnn8sp	os7m_tb__sp
os7mtbnn8d	os7m_tb__d
os7mtbnn8dp	os7m_tb__dp
os7mtbnn8dt	os7m_tb__dt
os7mtbnn8dpl	os7m_tb__dpl
Cortex M0, little endian	
os6mtlnn8xr	os6m_tl__xr
os6mtlnn8r	os6m_tl__r
os6mtlnn8s	os6m_tl__s
os6mtlnn8sp	os6m_tl__sp
os6mtlnn8d	os6m_tl__d
os6mtlnn8dp	os6m_tl__dp
os6mtlnn8dt	os6m_tl__dt
os6mtlnn8dpl	os6m_tl__dpl
Cortex M, big endian	
os6mtbnn8xr	os6m_tb__xr
os6mtbnn8r	os6m_tb__r
os6mtbnn8s	os6m_tb__s
os6mtbnn8sp	os6m_tb__sp
os6mtbnn8d	os6m_tb__d
os6mtbnn8dp	os6m_tb__dp
os6mtbnn8dt	os6m_tb__dt
os6mtbnn8dpl	os6m_tb__dpl

Table 3.3: Comparison of embOS library names for EWARM V5 and EWARM v6

Chapter 4

Compiler specifics

4.1 Standard system libraries

embOS for Cortex M cores and IAR compiler may be used with IAR standard libraries for most of all projects.

Heap management and file operation functions of the standard system libraries from the EWARM V5 are not reentrant and therefore require some precaution for mutual exclusion when these functions are used with embOS.
If non thread-safe functions are used from different tasks, embOS functions may be used to encapsulate these functions and guarantee mutal exclusion.
With EWARM 5 this mutal exclusion can be automated using a special initialization function during system setup. Using C++ projects with dynamic object creation will also require the special initialization procedure to activate thread safe system library functions.
The system libraries from the IAR EWARM V6 come with built in hook functions which enable thread safe calls of all system functions automatically when supported by the operating system.
embOS compiled for the IAR Embedded Workbench V6 is prepared to use these hook functions. Adding one or two source code modules which are delivered with embOS activates the automatic thread locking functionallity of the new IAR dlib V6.

4.2 Thread-safe system libraries

Using embOS with C++ projects and file operations or just normal call of heap management functions may require thread-safe system libraries if these functions are called from different tasks. Thread-safe system libraries require some locking mechanism which is RTOS specific.
Note that the locking mecanism differs according the IAR workbench version.
Using the new IAR workbench V6 requires the embOS libraries compiled for the workbench V6. To activate thread safe system library functionallity, one or two special source modules delivered with embOS have to be included in the project.
Using the older workbench V5 requires the embOS libraries compiled for the workbench V5 and need an initialization function which has to be called to enable thread safe system library access.

4.2.1 Thread safe system libraries with IAR compiler V6.x

The new runtime libraries of the IAR compiler / workbench implement hook functions for thread safe usage of system functions which can be used and supported with embOS.
To enable the automatic thread safe locking functions, the source module xmtx.c which is included in every CPU specific Setup folder of the embOS shipment has to be included in the project.
To support thread safe fil i/o functionallity, the source module xmtx2.c has to be added.
The embOS libraries compiled for and with the new IAR compiler / workbench V6 come with all code required to automatically handle the thread safe system libraries when the source module xmtx.c or xmtx2.c from the embOS shipment are included in the project.
Note that thread safe system library and file i/o support is required only, when non thread safe functions are called from multiple tasks, or thread local storage is used in multiple tasks.

4.2.2 Thread safe system libraries with IAR compiler V5.x

To switch the system libraries to thread safe mode, the embOS function OS_INIT_SYS_LOCKS() which is included in the embOS libraries compiled for the IAR Embedded Workbench V5 has to be called before the system is started. A typical embOS initialization for thread safe usage of system libraries with IAR workbench / compiler V5 would look like follows:

```
int main(void) {
  OS_IncDI();                       /* Initially disable interrupts  */
  OS_InitKern();                    /* Initialize OS                 */
  OS_INIT_SYS_LOCKS();              /* Activate thread safe library  */
  OS_InitHW();                      /* Initialize Hardware for OS    */
  /* 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;
}
```

4.3 Thread-Local Storage TLS

The new dlib of EWARM V6 supports usage of thread-local storage. Several library objects and functions need local variables which have to be unique to a thread.

Thread-local storage will be required when these functions are called from multiple threads.

embOS for EWARM V6 is prepared to support the tread-local storage, but does not use it per default. This has the advantage of no additional overhead as long as thread-local storage is not needed by the application.

The embOS implementation of thread-local storage allows activation of TLS separately for every task.

Only tasks that call functions using TLS need to activate the TLS by calling an initialization function when the task is started.

The IAR runtime environment allocates the TLS memory on the heap. Using TLS with multiple tasks shall therefore use thread safe system library functionallity which is automatically enabled when the *xmtx.c* module from the embOS distribution is added to the project.

Library objects that need thread-local storage when used in multiple tasks are:

- error functions -- errno, strerror.
- locale functions -- localeconv, setlocale.
- time functions -- asctime, localtime, gmtime, mktime.
- multibyte functions -- mbrlen, mbrtowc, mbsrtowc, mbtowc, wcrtomb, wcsrtomb, wctomb.
- rand functions -- rand, srand.
- etc functions -- atexit, strtok.
- C++ exception engine.

4.3.1 OS_ExtendTaskContext_TLS()

Description

OS_ExtendTaskContext_TLS() may be called from a task to initialize and use Thread-local storage.

Prototype

`void` OS_ExtendTaskContext_TLS(`void`)

Return value

None.

Additional Information

OS_ExtendTaskContext_TLS() shall be the first function called from a task when TLS should be used in the specific task.

The function must not be called multiple times from one task.

The thread-local storage is allocated on the heap. To ensure thread safe heap management, the thread safe system library functionallity shall also be enabled when using TLS.

Thread safe system library calls are automatically enabled when the source module *xmtx.c* which is delivered with embOS in the CPU specific Setup folders is included in the project.

The function is available in embOS for EWARM6 only.

Example

The following printout shows the sample application *Start_TLS.c* which demonstrates the usage of task specific TLS in an application.

The sample program is included in the CPU specific *Application* folders which contains all sample applications for embOS.

```
/*********************************************************************
*                                                                    *
*       (c) 1995 - 2010  SEGGER Microcontroller GmbH & Co KG         *
*                                                                    *
*       www.segger.com     Support: support@segger.com               *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
File    : Start_TLS.c
Purpose : Sample application to demonstrate usage of TLS
--------  END-OF-HEADER  ---------------------------------------------
*/

#include "RTOS.h"
#include <errno.h>
OS_STACKPTR int StackHP[128], StackLP[128];  /* Task stacks          */
OS_STACKPTR int StackMP[128];                /* Task stacks          */
OS_TASK TCBHP, TCBLP;                        /* Task-control-blocks */
OS_TASK TCBMP;                               /* Task-control-blocks */

static void HPTask(void) {
  OS_ExtendTaskContext_TLS();       // Initialize TLS for this task
  while (errno != 0) {
    // errno is local to this task, we should not arrive here
  }
  errno = 3;           // Simulate a task specific error
  while (1) {
    OS_Delay (10);
      while (errno != 3) {
        // errno is local to this task, we should not arrive here
      }
  }
}

static void MPTask(void) {
  OS_TLS_Init();       // Initialize TLS for this task
  while (errno != 0) {
    // errno is local to this task, we should not arrive here
  }
  errno = 2;           // Simulate a task specific error
  while (1) {
    OS_Delay (10);
    while (errno != 2) {
      // errno is local to this task, we should not arrive here
    }
  }
}

static void LPTask(void) {  // This task does not use TLS
  while (errno != 1) {
    // errno is not local to this task, we expect the global value
    // set in main() and should not arrive here
  }
  while (1) {
    OS_Delay (50);
  }
}


/*********************************************************************
*
*       main()
*
********************************************************************/
int main(void) {
  OS_IncDI();                      /* Initially disable interrupts  */
  errno = 1;                       /* Simulate an error             */
  OS_InitKern();                   /* Initialize OS                 */
  OS_InitHW();                     /* Initialize Hardware for OS    */
  /* You need to create at least one task here !                    */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBMP, "MP Task", MPTask,  70, StackMP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking            */
  return 0;
}
```

4.3.2 OS_ExtendTaskContext_TLS_VFP()

Description

`OS_ExtendTaskContext_TLS_VFP()` has to be called as first function in a task, when thread-local storage and thread safe floatingpoint processor support is needed in the task.

Prototype

`void` OS_ExtendTaskContext_TLS_VFP(`void`)

Return value

None.

Additional Information

`OS_ExtendTaskContext_TLS_VFP()` shall be the first function called from a task when TLS and VFP should be used in the specific task.
The function must not be called multiple times from one task.
The thread local storage is allocated on the heap. When the task is terminated, or terminates itself by a call of OS_TerminateTask() , the memory used for TLS is automatically freed and put back into the free heap memory.
To ensure thread safe heap management, the thread safe system library functionallity should also be enabled when using TLS.
Thread safe system library calls are automatically enabled when the source module `xmtx.c` delivered with embOS is included in the project.
The function `OS_ExtendTaskContext_TLS_VFP()` is available since embOS version V3.84 for EWARM6 or later only.
The task specific TLS management is generated as embOS task extension together with the storage needed for the VFP registers. The VFP registers are automatically saved onto the task stack when the task is suspended, and restored, when the task is resumed. Additional task extension by a call of `OS_ExtendTaskContext()` is impossible.
The function is available in all embOS libraries with VFP support:
os7m_tlv_xx or os7m_tbv_xx.

4.4 Vector Floating Point support VFPv4

Some Cortex M4 / M4F MCUs come with an integrated vectored floating point unit VFPv4.
When selecting the CPU and activating the VFPv4 support in the project options, the compiler and linker will add efficient code which uses the VFP when floating point operations are used in the application.
With embOS, the VFP registers have to be saved and restored when preemptive or cooperative task switches are performed.
For efficiency reasons, embOS does not save and restore the VFP registers for every task automatically. The context switching time and stack load are therefore not affected when the VFP unit is not used or needed.
Saving and restoring the VFP registers can be enabled for every task individually by extending the task context of the tasks which need and use the VFP.

4.4.1 OS_ExtendTaskContext_VFP()

Description

`OS_ExtendTaskContext_VFP()` has to be called as first function in a task, when the VFP is used in the task and the VFP registers have to be added to the task context.

Prototype

`void` OS_ExtendTaskContext_VFP(`void)`

Return value

None.

Additional Information

OS_ExtendTaskContext_VFP() extends the task context to save and restore the VFP registers during context switches.
Additional task context extension for a task by calling `OS_ExtendTaskContext()` is not allowed and will call the embOS error handler `OS_Error()` in debug builds of embOS.
There is no need to extend the task context for every task. Only those tasks using the VFP for calculation have to be extended.
When Thread-local Storage (TLS) is also needed in a task, the new embOS function `OS_ExtendTaskContext_TLS_VFP()` has to be called to extend the task context for TLS and VFP.

4.4.2 Using embOS libraries with VFP support

When VFP support is selected as project option, one of the embOS libraries with VFP support have to be used in the project.
these are named os7m_tlv_xx.a or os7m_tbv_xx.a
The embOS libraries for VFP support require that the VFP is switched on during startup and remains switched on during program execution.
When selecting the VFP support in the project options, the standard IAR startup code will automatically activate the VFP unit.
Using your own startup code, ensure that the VFP is switched on during startup.
When the VFP unit is not switched on, the embOS scheduler will fail.
The debug version of embOS checks whether the VFP is switched on when embOS is initailized by calling OS_InitKern().
When the VFP unit is not detected or not switched on, the embOS error handler OS_Error() is called with error code OS_ERR_CPU_STATE_ILLEGAL.

4.4.3 Using the VFP in interrupt service routines

Using the VFP in interrupt service routines requires additional functions to save and restore the VFP registers.
The implementation of VFP support in embOS disables the automatic context saving of VFP registers which is normally activated after reset.
embOS disables the VFP context saving feature of the Cortex M4F at all. This has the advantage that no additional stack is needed in tasks not using the VFP unit.

As the IAR compiler does not add additional code to save and restore the VFP registers on entry and exit of interrupt service routines, it is the users responsibility to save the VFP registers on entry of an interrupt service routine when the VFP is used in the ISR.
embOS delivers two functions to save and restore the VFP context in an interrupt service routine.

4.4.3.1 OS_VFP_Save()

Description

`OS_VFP_Save()` has to be called as first function in an interrupt service routine, when the VFP is used in the interrupt service routine. The function saves the temporary VFP registers on the stack.

Prototype

`void` OS_VFP_Save(`void`)

Return value

None.

Additional Information

OS_VFP_Save() declares a local variable which reserves space for all temporary floating point registers and stores the registers in the variable.
After calling the OS_VFP_Save() function, the interrupt service routine may use the VFP for calculation without destroying the saved content of the VFP registers.
To restore the registers, the ISR has to call OS_VFP_Restore() at the end.
The function may be used in any ISR regardless the priority. It is not restricted to low priority interrupt functions.

4.4.3.2 OS_VFP_Restore()

Description

`OS_VFP_Restore()` has to be called as last function in an interrupt service routine, when the VFP registers were saved by a call of OS_VFP_Save() at the beginning of the ISR. The function restores the temporary VFP registers from the stack.

Prototype

`void` OS_VFP_Restore(`void`)

Return value

None.

Additional Information

OS_VFP_Restore() restores the temporary VFP registers which were saved by a previous call of OS_VFP_Save().
It has to be used together with OS_VFP_Save() and should be the last function called in the ISR.

Example of a low priority interrupt service routine using VFP:

```
void ADC_ISR_Handler(void) {
  OS_VFP_Save();  // Save VFP registers
  OS_EnterInterrupt();
  DoSomeFloatOperation();
  OS_LeaveInterrupt();
  OS_VFP_Restore();  // Restore VFP registers.
}
```

In low priority interrupt service routines, OS_EnterInterrupt() is called to inform embOS that an interrupt handler is running and blocks task switches until OS_LeaveInterrupt() is called.
After calling OS_EnterInterrupt(), or OS_EnterNestableInterrupt(), any embOS function which is allowed to be called from an ISR may be called.

Example of a high priority interrupt service routine using VFP:

```
void ADC_ISR_Handler(void) {
  OS_VFP_Save();      // Save VFP registers
  DoSomeFloatOperation();
  OS_VFP_Restore();  // Restore VFP registers.
}
```

In interrupt service routines running at higher priority, no embOS functions except OS_VFP_Save() and OS_VFP_Restore may be called. Not even OS_EnterInterrupt().

Chapter 5

Stacks

5.1 Task stack for Cortex M

All embOS tasks execute in thread mode using the process stack pointer. The stack itself is located in any RAM location. Each task uses its individual stack. The stack poiter is initialized and set every time a task is activated by the scheduler. The stack-size required for a task is the sum of the stack-size of all routines plus a 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 calling embOS-routines.

For the Cortex M CPUs, this minimum basic task stack size is about 72 bytes. 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 one exception too. We recommend at least 256 bytes stack as a start.
When Cortex M4F CPUs with VFPv4 are used, keep in mind that the VFP registers may be pushed onto the stack and will require additional 136 bytes of stack.

5.2 System stack for Cortex M

The embOS system executes in thread mode, the scheduler executes in handler mode. The scheduler and OS_Idle() execute using the main stack pointer. 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.

The size of the system stack can be changed by modifying the CSTACK size in your *.icf file.
We recommend a minimum stack size of 256 bytes for the CSTACK.
Using a Cortex M4F CPU with floating point unit may require more stack, because the floating point registers might be saved on the system stack also.

5.3 Interrupt stack for Cortex M

If a normal hardware exception occurs, the Cortex M core switches to handler 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.
The main stack is also used as stack by the embOS scheduler and during idle times, when no task is ready to run and OS_Idle() is executed.

Chapter 6

Interrupts

The Cortex M core comes with an built in vectored interrupt controller which supports up to 32 separate interrupt sources. The real number of interrupt sources depends on the specific target CPU.

6.1 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 accepted and 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 stackand continues the interrupted function.

6.2 Defining interrupt handlers in C

Interrupt handlers for Cortex M cores are written as normal C-functions which do not take parameters and do not return any value. Interrupt handler which call an embOS function need a prolog and epilog function as described in the generic manual and in the examples below.

Example

Simple interrupt routine:

```
static void _Systick(void) {
  OS_EnterNestableInterrupt(); // Inform embOS that interrupt code is running
  OS_HandleTick();             // May be interrupted by higher priority interrupts
  OS_LeaveNestableInterrupt(); // Inform embOS that interrupt handler is left
}
```

6.3 Interrupt vector table

After Reset, the ARM Cortex M 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 handlers.

The interrupt vector table is located in a C source or assembly file 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. When using your own interrupt vector table, ensure that the addresses of the embOS exception handlers OS_Exception() and OS_Systick() are included.
When the vector table is not located at address 0x00, the vetor base register in the NVIC controller has to be initialized to point to the vector table base address.

6.4 Interrupt-stack switching

Since Cortex M core based controllers have two separate stack pointers, and embOS runs the user application on the process stack, there is no need for explicit stack-switching in an interrupt routine which runs on the main stack. The routines OS_EnterIntStack() and OS_LeaveIntStack() are supplied for source code compatibility to other processors only and have no functionality.

6.5 Fast interrupts

6.5.1 Fast interrupts with Cortex M0

Since there is no posibility to read or write the current interrupt priority embOS for Cortex M0 does not support Fast interrupts.

6.5.2 Fast interrupts with Cortex M3 / M4 and M4F

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 numbers define a higher priority. All interrupts with priority level from 0 to 127 are never disabled. These interrupts are named Fast interrupts. You must not execute any embOS function from within a fast interrupt function.

6.6 Interrupt priorities

This chapter describes interrupt priorities supported by the CortexM CPU cores.
The priority is any number between 0 and 255 as seen by the CPU core. With embOS and its own setup functions for the interrupt controller und priorities, there is no difference in the priority values regardless the different preemption level of specific devices.
Using the CMSIS functions to setup interrupt priorities requires different values for the priorities. These values depend on the number of preemtion levels of the specific chip. a description is found in the chapter CMSIS.

6.6.1 Interrupt priorities with Cortex M0 core

The Cortex M0 supports up to 4 levels of programmable priority. Every interrupt with a higher preemption level may preempt any other interrupt handler running on a lower preemption level. Interrupts with equal preemption level may not preempt each other.

6.6.2 Interrupt priorities with Cortex M3 / M4 and M4F cores

The Cortex M3 / M4 and M4F support up to 256 levels of programmable priority with a maximum of 128 levels of preemption. Most Cortex M chips have fewer supported levels, for example 8, 16, 32, and so on. The chip designer can customize the chip to obtain the levels required. At least, there is a minimum of 8 preemption levels. Every interrupt with a higher preemption level may preempt any other interrupt handler running on a lower preemption level. Interrupts with equal preemption level may not preempt each other.

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

6.6.3 Priority of the embOS scheduler

The embOS scheduler runs on the lowest interrupt priority. The scheduler may be preempted by any other interrupt with higher preemption priority level. The application interrupts shall run on higher preemption levels to ensure short reaction time.

During initialization, the priority of the embOS scheduler is set to 0x03 for Cortex M0 and to 0xFF for Cortex M3 / M4 and M4F, which is the lowest preemption priority regardless of the number of preemption levels.

6.6.4 Priority of the embOS system timer

The embOS system timer runs on the second lowest preemption level. Thus, the embOS timer may preempt the scheduler. Application interrupts which require fast reaction should run on a higher preemption priority level.

6.6.5 Priority of embOS software timers

The embOS software timer callback functions are called from the scheduler and run on the schedulers preemption priority level which is the lowest interrupt priority level. To ensure short reaction time of other interrupts, other interrupts should run on a higher preemption priority level and the software timer callback functions should be as short as possible.

6.6.6 Priority of application interrupts for Cortex M0 core

Application interrupts may run on any priority level between 0 to 3. However, interrupts, which require fast reaction should run on higher priority levels than the embOS scheduler and the embOS system timer to allow preemption of theses interrupt handlers. We recommend that application interrupts should run on a higher preemption level than the embOS scheduler, at least at the second lowest preemption priority level.

6.6.7 Priority of application interrupts for Cortex M3 / M4 core

Application interrupts using embOS functions may run on any priority level between 255 to 128. However, interrupts, which require fast reaction should run on higher priority levels than the embOS scheduler and the embOS system timer to allow preemption of theses interrupt handlers. Interrupt handler which require fastest reaction may run on higher priorities than 128, but must not call any embOS function (->Fast interrupts). We recommend that application interrupts should run on a higher preemption level than the embOS scheduler, at least at the second lowest preemption priority level.

As the number of preemption levels is chip specific, the second lowest preemption priority varies depending on the chip. If the number of preemption levels is not documented, the second lowest preemption priority can be set as follows, using embOS functions:

```
unsigned char Priority;
OS_ARM_ISRSetPrio(_ISR_ID, 0xFF);            // Set to lowest level, ALL BITS set
Priority  = OS_ARM_ISRSetPrio(_ID_TICK, 0xFF); // Read priority back
Priority -= 1;                               // Lower preemption level
OS_ARM_ISRSetPrio(_ISR_ID, Priority);
```

6.6.8 Priority grouping for Cortex M3 / M4 core

The number of preemption levels may be limited by programming the priority group level in the application interrupt and reset control register of the chip. embOS does not modify this register, thus allowing the maximum number of preemption levels which are implemented by the chip design. It is recommended, not to change the priority grouping setting.

6.7 Interrupt nesting

The Cortex M CPU uses a priority controlled interrupt scheduling which allows nesting of interrupts per default. Any interrupt or exception with a higher preemption priority may interrupt an interrupt handler running on a lower preemption priority. An interrupt handler calling embOS functions has to start with an embOS prolog function that informs embOS that an interrupt handler is running. For any interrupt handler, the user may decide individually whether this interrupt handler may be preempted or not by choosing the prolog function.

6.7.1 OS_EnterInterrupt()

Description

OS_EnterInterrupt(), disables nesting

Prototype

```
void OS_EnterInterrupt(void)
```

Return value

None.

Additional Information

OS_EnterInterrupt() has to be used as prolog function, when the interrupt handler should not be preempted by any other interrupt handler that runs on a priority below the fast interrupt priority. An interrupt handler that starts with OS_EnterInterrupt() has to end with the epilog function OS_LeaveInterrupt().

Example

Interrupt-routine that can not be preempted by other interrupts

```
static void _Systick(void) {
  OS_EnterInterrupt();// Inform embOS that interrupt code is running
  OS_HandleTick();    // Can not be interrupted by higher priority interrupts
  OS_LeaveInterrupt();// Inform embOS that interrupt handler is left
}
```

6.7.2 OS_EnterNestableInterrupt()

Description

OS_EnterNestableInterrupt(), enables nesting

Prototype

```
void OS_EnterNestableInterrupt(void)
```

Return value

None.

Additional Information

OS_EnterNestableInterrupt(), allow nesting. OS_EnterNestableInterrupt() may be be used as prolog function, when the interrupt handler may be preempted by any other interrupt handler that runs on a higher interrupt priority. An interrupt handler that starts with OS_EnterNestableInterrupt() has to end with the epilog function OS_LeaveNestableInterrupt().

Example

Interrupt-routine that can be preempted by other interrupts

```
static void _Systick(void) {
  OS_EnterNestableInterrupt();// Inform embOS that interrupt code is running
  OS_HandleTick();            // Can be interrupted by higher priority interrupts
  OS_LeaveNestableInterrupt();// Inform embOS that interrupt handler is left
}
```

6.7.3 Required embOS system interrupt handler

embOS for Cortex M core needs two exception handler which belong to the system itself. Both are delivered with embOS. Ensure that they are referenced in the vector table.

6.7.3.1 OS_Exception(), PendSV_Handler(), the scheduler entry

OS_Exception() is the scheduler entrance of embOS. It runs on the lowest interrupt priority. Whenever scheduling is required, this exception is triggered by embOS. OS_Exception() has to be called by the PendSV exception of the Cortex M CPU. Ensure that the address of OS_Exception() is inserted in the vector table at the correct position.
As embOS for Cortex M is fully CMSIS compliant, the CMSIS generic PendSV handler name PendSV_Handler may be used in the vector table. It is automatically mapped to the OS_Exception() handler of embOS.
The vector tables which come with embOS are already setup and should be used and modified for the application.

6.7.3.2 OS_Systick(), SysTick_Handler(), the embOS system timer handler

OS_Systick() is the interrupt handler which manages the system timer. The system timer is initialized during OS_InitHW(). The embOS system timer uses the SYSTICK timer of the Cortex M CPU and runs on a low preemption priority level which is one level higher than the lowest preemption priority level. Ensure that the address of OS_Systick() is inserted in the vector table at the correct position.
As embOS for Cortex M is fully CMSIS compliant, the CMSIS generic systick handler name SysTick_Handler may be used in the vector table. It is automatically mapped to the OS_Systick() handler in the RTOSInit source files of the specific CPU.
The vector tables which come with embOS are already setup and should be used and modified for the application.

6.8 Interrupt handling with vectored interrupt controller

For the Cortex M core, 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:

6.8.1 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	Description
ISRIndex	Index of the interrupt source which should be enabled. Note that the index counts from 0 for the first entry in the vector table.

Table 6.1: OS_EnterInterrupt() parameter list

Return value

None.

Additional 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.
Note that the ISRIndex counts from 0 for the first entry in the vector table.
The first peripheral index therefore has the ISRIndex 16, because the first peripheral interrupt vector is located after the 16 generic vectors in the vector table.
This differs from index values used with CMSIS.

6.8.2 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	Description
ISRIndex	Index of the interrupt source which should be disabled. Note that the index counts from 0 for the first entry in the vector table.

Table 6.2: OS_EnterInterrupt() parameter list

Return value

None.

Additional 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.
Note that the ISRIndex counts from 0 for the first entry in the vector table.

The first peripheral index therefore has the ISRIndex 16, because the first peripheral interrupt vector is located after the 16 generic vectors in the vector table.
This differs from index values used with CMSIS.

6.8.3 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	Description
ISRIndex	Index of the interrupt source which should be modified. Note that the index counts from 0 for the first entry in the vector table.
Prio	The priority which should be set for the specific interrupt. Prio ranges from 0 (highest priority) to 255 (lowest priority)

Table 6.3: OS_EnterInterrupt() parameter list

Return value

None.

Additional 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.
Note that the ISRIndex counts from 0 for the first entry in the vector table.
The first peripheral index therefore has the ISRIndex 16, because the first peripheral interrupt vector is located after the 16 generic vectors in the vector table.
This differs from index values used with CMSIS.
The priority value is independent from the chip specific preemption levels. Any value between 0 and 255 can be used, were 255 always is the lowest priority and 0 is the highest priority.
The function can be called to set the priority for all interrupt sources, regardless embOS is used in the specified interrupt handler.
Note that interrupt handler running on priorities from 127 or higher must not call any embOS function.

6.8.4 Interrupt identifier ISRIndex with embOS functions

The embOS functions for NVIC setup require an peripheral identifier index (ISRIndex) to address the peripheral.
The identifier number may differ from the ID values used by vendor specific header files and functions, it also differs from the index numbers used with CMSIS.
Using the embOS functions requires an index counting from 0 for the first entry in the interrupt vector table.
The first 16 vectors (index 0 to 15) address Cortex system interrupt sources.
The first peripheral vector is on the 16th place in the vactor table and has to be addressed with id index 16 when using the embOS functions to control the priority or enable and disable the interrupt in the NVIC controller.

6.8.5 Interrupt priority values with embOS functions

The interrupt priority values used with the embOS functions alwasy range from 0 to 255 regardless the number of preemption levels of the specific CPU.
The interrupt priority value is written into the specified control registes as is.

255 is the lowest interrupt priority level.
0 is the highest interrupt priority level.
All interrupt handler using embOS functions have to be setup with priorities between 128 and 255 (low priority level).
All interrupt handler running on higher priorties between 0 and 127 must not call any embOS functions, but can be setup, enabled and disabled, using the embOS functions described above.
Note that CMSIS uses different values to setup the interrupt priorities. The values used with CMSIS depend on the number of preemption levels that the specifc CPU supports.

6.8.6 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.

Chapter 7

CMSIS

ARM introduced the Cortex Microcontroller Software Interface Standard (CMSIS) as a vendor independent hardware abstraction layer for simplifying software re-use.

The standard enables consistent and simple software interfaces to the processor, for peripherals, for real time operating systems as embOS and other middleware.

As SEGGER is one of the CMSIS partners, embOS for Cortex M is fully CMSIS compliant.

embOS comes with a generic CMSIS start project which should run on any Cortex M3 CPU. All other start projects, even those not based on CMSIS, are also fully CMSIS compliant and can be used as starting points for CPU specific CMSIS projects.

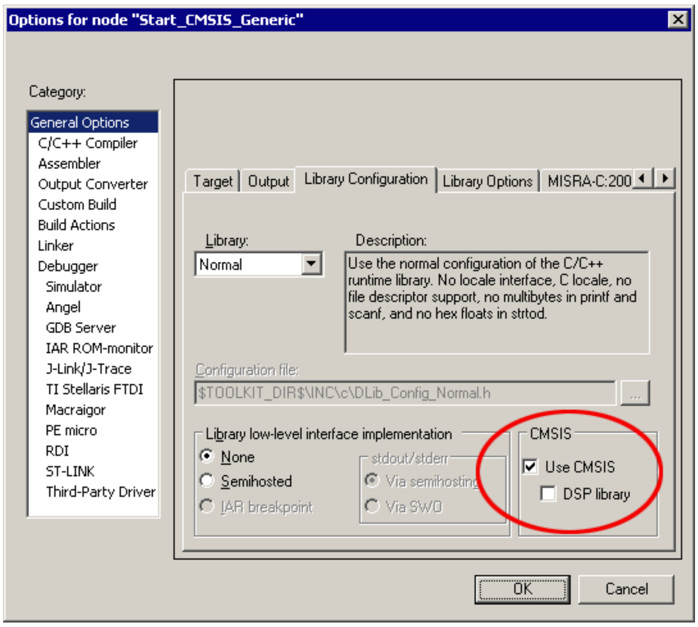
How to use the generic project and adding vendor specific files to this or other projects is explained in the following chapters.

7.1 CMSIS with IAR EWARM V6

Since version 6.10.5, the IAR Embedded Workbench comes with the option to activate CMSIS support in the project settings.
All CMSIS generic header files are delivered with the IAR Embedded Workbench and the include paths are automatically set, when CMSIS support is activated in the General Options / Library Configuration.

Using CMSIS support as project option is required with IAR EWARM V6.20 or later when a CMSIS based project shall be used.
The worbench delivers all core specific header and source files.
It is required, that these files are delivered from the ide and are NOT part of the project.
To use the CMSIS option of the workbench with a CMSIS based embOS project, proceed as follows:

- Remove the CoreSupport subfolder of the embOS project folders, or rename all files found in that folder if the CoreSupport folder exists.
- Activate the CMSIS support in the project options by checking the "Use CMSIS" option.



Options for node "Start_CMSIS_Generic"

Category: General Options

Library: Normal

Description: Use the normal configuration of the C/C++ runtime library. No locale interface, C locale, no file descriptor support, no multibytes in printf and scanf, and no hex floats in strtod.

Configuration file: $TOOLKIT_DIR$\INC\c\DLib_Config_Normal.h

Library low-level interface implementation: ● None ○ Semihosted ○ IAR breakpoint

stdout/stderr: ● Via semihosting ○ Via SWO

CMSIS: ☑ Use CMSIS □ DSP library

7.2 The generic CMSIS start project

The folder Start\BoardSupport\CMSIS contains a generic CMSIS start project that should run on any Cortex M3 / M4 / M4F core.

The subfolder DeviceSupport\ contains the device specific source and header files which have to be replaced by the device specific files of the CM3 / CM4 vendor to make the CMSIS sample start project device specific.

7.3 Device specific files needed for embOS with CMSIS

- ***Device*.h**: Contains the device specific exception and interrupt numbers and names. embOS needs the Cortex M3 generic exception names PendSV_IRQn and SysTick_IRQn only which are vendor independent and common for all devices. The sample file delivered with embOS does not contain any peripheral interrupt vector numbers and names as those are not needed by embOS.
  To make the embOS CMSIS sample device specific and allow usage of peripheral interrupts, this file has to be replaced by the one which is delivered from the CPU vendor.
- **System_*Device*.h**: Declares at least the two required system timer functions which are used to initialize the CPU clock system and one variable which allows the application software to retrieve information about the current CPU clock speed. The names of the clock controlling functions and variables are defined by the CMSIS standard and are therefore identical in all vendor specific implementations.
- **System_*Device*.c**: Implements the core specific functions to initialize the CPU, at least to initialize the core clock. The sample file delivered with embOS contains empty dummy functions and has to be replaced by the vendor specific file which contains the initialization functions for the core.
- **Startup_*Device*.s**: The startup file which contains the initial reset sequence and contains exception handler and peripheral interrupt handler for all interrupts.
  The handler functions are declared weak, so they can be overwritten by the application which implements the application specific handler functionality.
  The sample which comes with embOS only contains the generic exception vectors and handler and has to be replaced by the vendor specific startup file.

Startup code requirements:

The reset handler HAS TO CALL the **SystemInit()** function which is delivered with the core specific system functions.
When using a Cortex M4 or M4F CPU which may have a VFPv4 floating point unit equipped, please ensure that the reset handler activates the VFP when VFPv4 is selected in the project options.
When VFP-support is not selected, the VFP should not be switched on.
The original IAR startup code from the IAR runtime library ensures that activation of the VFP depends on the project settings.
Otherwise, the SystemInit() function delivered from the device vendor should also honor the project settings and enable the VFP or keep it disabled according the project settings.
Using CMSIS compliant startup code from the chip vendors may require modification if it enables the VFP unconditionally.

7.4 Device specific functions/variables needed for embOS with CMSIS

The embOS system timer is triggered by the Cortex M generic system timer. The correct core clock and pll system is device specific and has to be initialized by a low level init function called from the startup code.
embOS calls the CMSIS function *SysTick_Config()* to set up the system timer. The function relies on the correct core clock initialization performed by the low level initialization function *SystemInit* and the value of the core clock frequency which has to be written into the *SystemCoreClock* variable during initialization or after calling *SystemCoreClockUpdate()*.

- **SystemInit()**:The system init function is delivered by the vendor specific CMSIS library and is normally called from the reset handler in the startup code. The system init function has to initialize the core clock and has to write the CPU frequency into the global variable *SystemCoreClock*.
- **SystemCoreClock**: Contains the current system core clock frequency and is initialized by the low level initialization function *SystemInit()* during startup. embOS for CMSIS relies on the value in this variable to adjust its own timer and all time related functions.

Any other files or functions delivered with the vendor specific CMSIS library may be used by the application, but are not required for embOS.

7.5 CMSIS generic functions needed for embOS with CMSIS

The embOS system timer is triggered by the Cortex M generic system timer which has to be initialized to generate periodic interrupts in a specified interval. The configuration function *SysTick_Config()* for the system timer relies on correct initialization of the core clock system which is performed during startup.

- **SystemCoreClockUpdate**: This CMSIS function has to update the *SystemCoreClock* variable according the current system timer initialization. The function is device sepcific and may be called before the *SystemCoreClock* variable is accessed or any function which relies on the correct setting of the system core clock variable is called. embOS calls this function during the hardware initialization function *OS_InitHW()* before the system timer is initialized.
- **SysTick_Config**: This CMSIS generic function is declared an implemented in the core_cm3.h file. It initializes and starts the SysTick counter and enables the SysTick interrupt. For embOS it is recommended to run the SysTick interrupt at the second lowest preemption priority. Therefore, after calling the *SysTick_Config()* function from *OS_InitHW()*, the priority is set to the second lowest preemption priority ba a call of *NVIC_SetPriority()*.
  The embOS function *OS_InitHW()* has to be called after initialization of embOS during main and is implemented in the *RTOSInit_CMSIS.c* file.
- **SysTick_Handler**: The embOS timer interrupt handler, called periodically by the interrupt generated from the SysTick timer. The SysTick_Handler is declared weak in the CMSIS startup code and is replaced by the embOS Systick_Handler function implemented in RTOSInit_CMSIS.c which comes with the embOS start project.
- **PendSV_Handler**: The embOS scheduler entry function. It is declared weak in the CMSIS startup code and is replaced by the embOS internal function contained in the embOS library. The embOS initialization code enables the *PendSV* exception and initializes the priority. The application **MUST NOT** change the *PendSV* priority.

7.6 Customizing the embOS CMSIS generic start project

The embOS CMSIS generic start project should run on every Cortex M3 / M4 or M4F CPU. As the generic device specific functions delivered with embOS do not initialize the core clock system and the pll, the timing is not correct, a real CPU will run very slow.

To run the sample project on a specific Cortex M3 / M4 / M4F CPU, replace all files in the *DeviceSupport\* folder by the versions delivered by the CPU vendor. The vendor and CPU specific files should be found in the CMSIS release package, or are available from the core vendor.

No other changes are necessary on the start project or any other files.

To run the generic CMSIS start project on a Cortex M0, you have to replace the embOS libraries by libraries for Cortex M0 and have to add Cortex M0 specific vendor files.

7.7 Adding CMSIS to other embOS start projects

All CPU specific start projects are fully CMSIS compatible. If required or wanted in the application, the CMSIS files for the specific CPU may be added to the project without any modification on existing files.
Note that the `OS_InitHW()` function or `__low_level_init()` in the RTOSInit file initialize the core clock system and pll of the specific CPU. The system clock frequency and core clock frequency are defined in the RTOSInit file.
If the application needs access to the `SystemCoreClock`, the core specific CMSIS startup code and core specific initialization function *SystemInit* has to be included in the project.
In this case, the `__low_level_init()` function and the `OS_InitHW()` function in RTOSInit may be replaced, or the CMSIS generic `RTOSInit_CMSIS.c` file may be used in the project.

7.7.1 Differences between embOS projects and CMSIS

Several embOS start projects are not based on CMSIS but are fully CMSIS compliant and can be mixed with CMSIS libraries from the device vendors.
Switching from embOS to CMSIS, or mixing embOS with CMSIS functions is possible without problems, but may require some modification when the interupt controller setup functions from CMSIS shall be used instead of the embOS functions.

7.7.1.1 Different peripheral ID numbers

Using CMSIS, the peripheral IDs to setup the interrupt controller start from 0 for the first peripheral interrupt. With emboS, the first peripheral is addressed with ID number 16.
embOS counts the first entry in the interrupt vector table from 0, so, the first peripheral interrupt following the 16 Cortex system interrupt entries, is 16.
When the embOS functions should be replaced by the CMSIS functions, this correction has to be taken into account, or if avaialable, the symbolic peripheral id numbers from the CPU specific CMSIS device header file may be used with CMSIS.
Note that using these IDs with the embOS functions will work only, when 16 is added to the IDs from the CMSIS device header files.

7.7.1.2 Different interrupt priority values

Using embOS functions, the interrupt priority value ranges from 0 to 255 and is written into the NVIC control registers as is, regardless the number of priority bits.
255 is the lowest priority, 0 is the highest priority.
Using CMSIS, the range of interrupt priority levels used to setup the interrupt controller depends on the number of priority bits implemented in the specific CPU.
The number of priority bits for the specific device shall be defined in the device specific CMSIS header file as `__NVIC_PRIO_BITS`.
If it is not defined in the device specifc header files, a default of 4 is set in the generic CMSIS core header file.
A CPU with 4 priority bits supports up to 16 preemption levels.
With CMSIS, the range of interrupt priorities for this CPU would be 0 to 15, where 0 is the highest priority and 15 is the lowest.
To convert an embOS priority value into a value for the CMSIS functions, the value has to be shifted to the right by (8 - `__NVIC_PRIO_BITS`).
To convert an CMSIS value for the interrupt priority into the value used with the embOS functions, the value has to be shifted to the left by (8 - `__NVIC_PRIO_BITS`).
In any case, half of the priorities with lower values (from zero) are high priorities which must not be used with any interrupt handler using embOS functions.

7.8 Interrupt and exception handling with CMSIS

The embOS CPU specific projects come with CPU specific vector tables and empty exception and interrupt handlers for the specific CPU. All handlers are named according the names of the CMSIS device specific handlers and are declared weak and can be replaced by an implementation in the application source files.

The CPU specific vector table and interrupt handler functions in the embOS start projects can be replaced by the CPU specific CMSIS startup file of the CPU vendor without any modification on other files in the project.

embOS uses the two Cortex M generic exceptions PendSV and SysTick and delivers its own handler functions to handle these exceptions.

All peripheral interrupts are device specific and are not used with embOS except for profiling support and system analysis with embOSView using a UART.

7.9 Enable and disable interrupts

The generic CMSIS functions `NVIC_EnableIRQ()` and `NVIC_DisableIRQ()` can be used instead of the embOS functions `OS_ARM_EnableISR()` and `OS_ARM_DisableISR()` functions which are implemented in the CPU specific RTOSInit files delivered with embOS.
Note that the CMSIS functions use different peripheral ID indices to address the specic interrupt number.
embOS counts from 0 for the first entry in the interrupt vector table, CMSIS counts from 0 for the first peripheral interrupt vector, which is ID number 16 for the embOS functions.
About these differences, also read chapter 7.7.1
To enable and disable interrupts in general, the embOS functions OS_IncDI() and OS_DecRI() or other embOS functions described in the generic embOS manual should be used instead of the intrinsic functions from the CMSIS library.

7.10 Setting the Interrupt priority

With CMSIS, the CMSIS generic function `NVIC_SetPriority()` can be used instead of the `OS_ARM_ISRSetPrio()` function which is implemented in the CPU specific RTOSInit files delivered with embOS.
Note that with the CMSIS function, the range of valid interrupt priority values depends on the number of priority bits defined and implemented for the specific device.
The number of priority bits for the specific device shall be defined in the device specific CMSIS header file as `__NVIC_PRIO_BITS`.
If it is not defined in the device specifc header files, a default of 4 is set in the generic CMSIS core header file.
A CPU with 4 priority bits supports up to 16 preemption levels.
With CMSIS, the range of interrupt priorities for this CPU would be 0 to 15, where 0 is the highest priority and 15 is the lowest.
About interrupt priorities in an embOS project, read chapter 6.5 and 6.6., about the differences between interrupt priority and ID values used to setup the NVIC controller, read chapter 7.7.1

Chapter 8

Using embOSView with Cortex M

The embOS profiling and analysis tool embOSView has been modified to be used with any Cortex M0 or M3 / M4 or M4F core together with J-Link. using a new communication channel which is available since embOS version 3.82g for Cortex M and implemented for Cortex M0 since version 3.82m. The previous communication method using a UART is still available. The CPU specific projects come with code for UART support as in previous versions, but use the new communication channel per default. The new communication channel does not rely on any peripherals and is therefore available for all Cortex M0 cores without the need of any peripheral or peripheral interrupt handler functions and runs on the CMSIS generic sample also.
embOSView is delivered with embOS and is described in the embOS generic manual.

8.1 Enable communication to embOSView

The communication to embOSView can be enabled by the compile time switch `OS_VIEW_ENABLE` which may be defined in the project settings or in the configuration file `OS_Config.h`.
If `OS_VIEW_ENABLE` is defined unequal to 0, the communication is enabled. In the RTOSInit files the `OS_VIEW_ENABLE` switch is set to 1 if not defined as project option.
The OS_Config.h file sets the compile time switch `OS_VIEW_ENABLE` to 0 when `DEBUG` is defined as 0.
Therefore, in the embOS start projects, the communication is enabled per default when using the `DEBUG` configurations, and is disabled when using the Release configurations.

8.2 Select the communication channel in the start project

When the communication to embOSView is enabled by setting the compile time switch OS_VIEW_ENABLE, the communication can be handled via UART or the new memory based communication channel using J-Link.
Since version 3.82g of embOS for Cortex M, the communication channel using J-Link. is activated per default in the embOS start projects.

8.2.1 Select a UART for communication

Set the compile time switch OS_VIEW_USE_UART unequal to 0 by project option/compiler preprocessor or in the OS_Config.h file to switch the communication from J-Link to UART.
In the RTOSInit files delivered with embOS, this switch is set to 0 if not defined by compiler preprocessor/project option or OS_Config.h.
OS_VIEW_ENABLE has to be set unequal to 0 to enable communication.

8.2.2 Select J-Link for communication

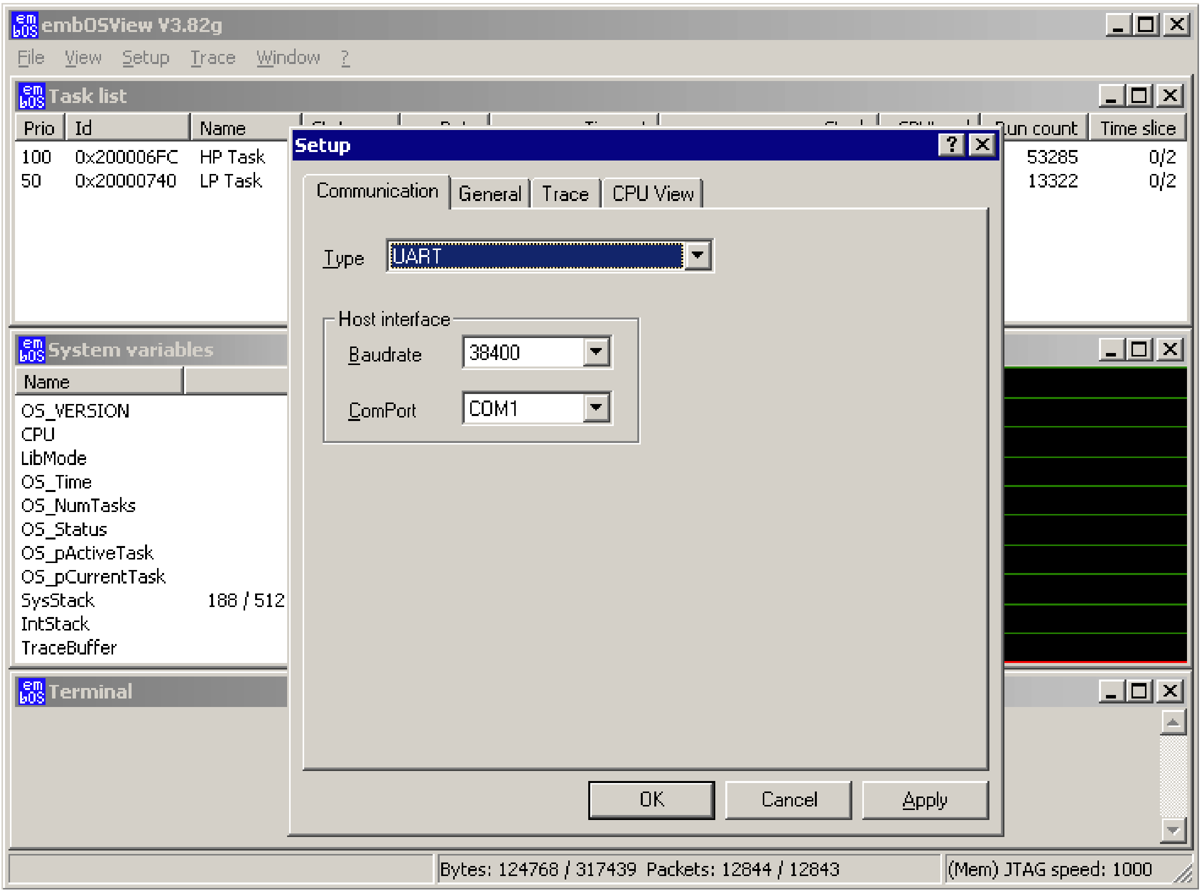
Per default, J-Link is selected as communication device in the embOS start projects. The compile time switch OS_VIEW_USE_UART is predefined to 0 in the CPU specific RTOSInit files, thus selecting the J-Link communication channel when not overwritten by project / compiler preprocessor options or in OS_Config.h.
OS_VIEW_ENABLE has to be set unequal to 0 to enable communication.

8.3 Setup embOSView for communication

When the communication to embOSView is enabled in the target application, embOSView can be used to analyze the running application.
The communication channel of embOSView has to be setup according the communication channel which was selected in the project.

8.3.1 Select a UART for communication

Start embOSView and chose menu Setup:



Prio	Id	Name	Run count	Time slice
100	0x200006FC	HP Task	53285	0/2
50	0x20000740	LP Task	13322	0/2

Name	
OS_VERSION	
CPU	
LibMode	
OS_Time	
OS_NumTasks	
OS_Status	
OS_pActiveTask	
OS_pCurrentTask	
SysStack	188 / 512
IntStack	
TraceBuffer	

Type: UART

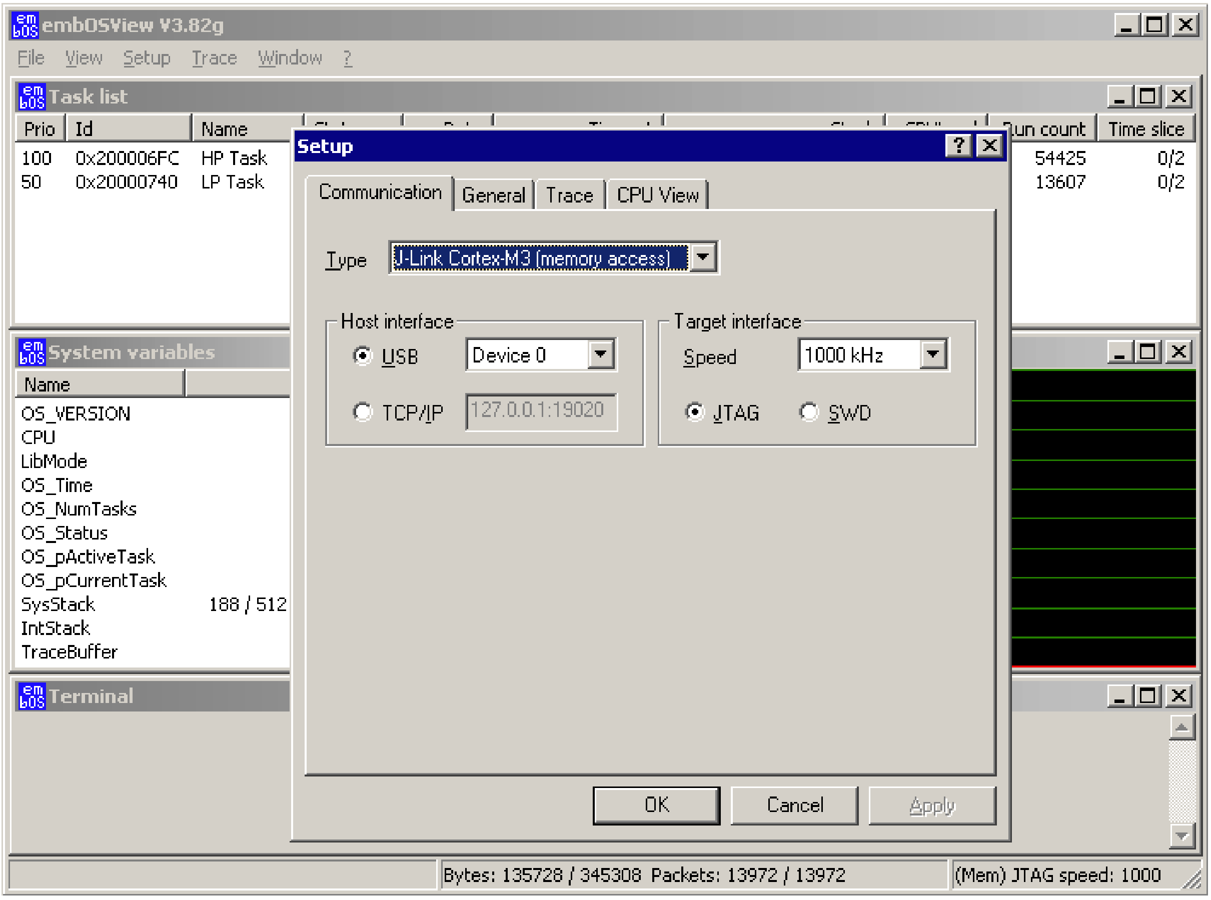
Host interface
Baudrate: 38400
ComPort: COM1

Bytes: 124768 / 317439 Packets: 12844 / 12843 (Mem) JTAG speed: 1000

In the Communication TAB chose UART in the Type selection listbox.
In the Host interface box select the Baudrate for communication and the COM port of the PC which should be connected to the target board.
The default baudrate of all projects is 38400 kBaud. The COM port list box lists all COM ports of the PC which are currently available.
The serial communication via UART is available in the target application if the project was compiled with the settings `OS_VIEW_USE_UART` unequal to 0 and `OS_VIEW_ENABLE` set unequal to 0.
The serial communication will work when the target is running stand alone or during a debug session, when the target is connected to the Debugger via JLink.
The serial connection can be used when the target board has a spare UART port and the OS_UART functions are enabled and included in the application.

8.3.2 Select J-Link for communication

embOS for Cortex M3 since version 3.82g supports a new communication channel to embOSView which uses J-Link to communicate with the running application. embOSView version 3.82g or higher and a J-Link-DLL is required to use a J-Link for communication.
To select this communication channel, start embOSView and open the Setup menu:



Prio	Id	Name	Run count	Time slice
100	0x200006FC	HP Task	54425	0/2
50	0x20000740	LP Task	13607	0/2

Name	
OS_VERSION	
CPU	
LibMode	
OS_Time	
OS_NumTasks	
OS_Status	
OS_pActiveTask	
OS_pCurrentTask	
SysStack	188 / 512
IntStack	
TraceBuffer	

Type: J-Link Cortex-M3 (memory access)

Host interface
● USB Device 0
○ TCP/IP 127.0.0.1:19020

Target interface
Speed 1000 kHz
● JTAG ○ SWD

Bytes: 135728 / 345308 Packets: 13972 / 13972 | (Mem) JTAG speed: 1000

In the Communication TAB chose J-Link Cortex-M3 (memory access) in the Type selection listbox.
In the Host interface box select the USB or TCP/IP channel which is used to communicate to your J-Link.
In the Target interface box select the communication speed of the target interface and the physical target connection which may be a JTAG or SWD connection.
The communication via J-Link is available in the target application if the project was compiled with the settings `OS_VIEW_USE_UART` equal to 0 and `OS_VIEW_ENABLE` set unequal to 0.

8.3.3 Use J-Link for communication and debugging in parallel

J-Link can be used to communicate with embOSView during a running debug session that uses the same J-Link as debug probe. To avoid problems, the target interface settings for J-Link should be the same in the debugger settings and in the embOSView Target interface settings. To use embOSView during a debug session, proceed as follows:

- Examine the target interface settings in the Debugger settings of the project.
- Before starting the debugger, start embOSView and set the same target interface as found in the debugger settings, for example SWD.
- Close embOSView
- Start the debugger
- Restart embOSView

J-Link will now communicate with the debugger and embOSView will communicate with embOS via J-Link as long as the application is running.

8.3.4 Restrictions for using J-Link with embOSView

The J-Link communication with the current version of embOSView can only be used when the vector table of the target application is located at address 0x00.

Chapter 9

STOP / WAIT Mode

9.1 Introduction

In case your controller does support some kind of power saving mode, it should be possible to use it also with embOS, as long as the timer keeps working and timer interrupts are processed. To enter that mode, you usually have to implement some special sequence in the function `OS_Idle()`, which you can find in embOS module `RTOSInit.c`.
Per default, the `wfi` instruction is executed in `OS_Idle()` to put the CPU into a low power mode.

Chapter 10

Technical data

10.1 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 minimum ROM requirement for the kernel itself is about 2.500 bytes.

In the table below, which is for release build, you can find minimum RAM size requirements for embOS resources. Note that the sizes depend on selected embOS library mode.

embOS resource	**RAM [bytes]**
Task control block	32
Resource semaphore	8
Counting semaphore	4
Mailbox	20
Software timer	20

Table 10.1: embOS memory requirements

Chapter 11

embOS C-Spy plug-in

This chapter gives a short overview about the embOS C-Spy plug-in for IAR Embedded Workbench®.

11.1 Overview

11.1.1 embOS C-Spy plug-in

SEGGER's embOS C-Spy plug-in for IAR Embedded Workbench provides embOS-awareness during debugging sessions. This enables you to inspect the state of several embOS primitives such as the task list, resource semaphores, mailboxes, and timers.

Since embOS version 3.62, you can check the general-purpose registers and inspect the call stack of all available application tasks.

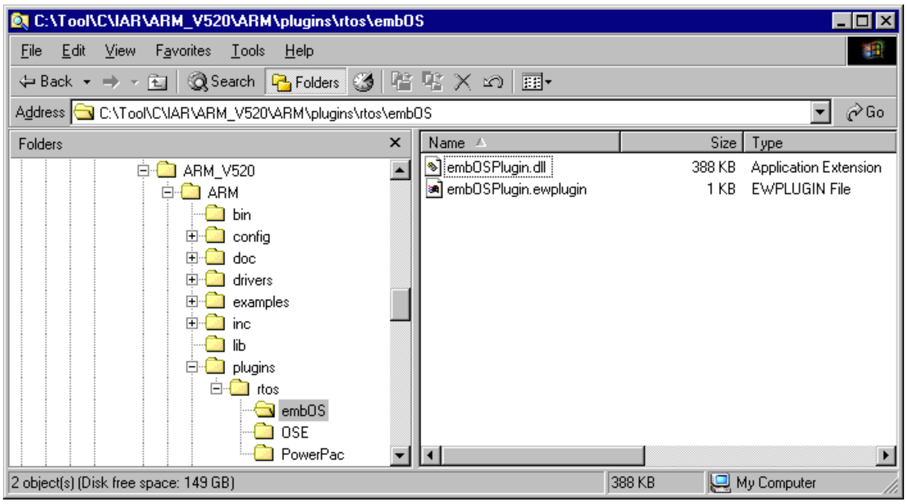
11.1.2 Requirements

To use the embOS C-Spy plug-in you need a version of IAR Embedded Workbench installed and a debug target which uses embOS. Specifically:

- An embOS version 3.62 or higher is required for complete compatibility. Older embOS versions use different internal structures and the C-Spy plug-in is therefore of limited use with version prior to 3.62.
- An IAR Embedded Workbench IDE with a C-SPY debugger version 5.x or higher is required for complete compatibility with Cortex M0 and M3.
- IAR Embedded Workbench IDE with a C-SPY debugger version 6.x or higher with the embOS plugin V6.0.6.7 or newer is required to support full task aware all-stack with CortexM4F and VFP support.
  Older plugins still work with a Cortex M4F with VFP, but can not show the task sensitive callstack.

11.2 Installation

The installation procedure is very straightforward because it only requires you to copy the contents of the embOS C-Spy plug-in package into the IAR CPU specific plug-in folder for `rtos` plug-ins. The directory structure may look like this:



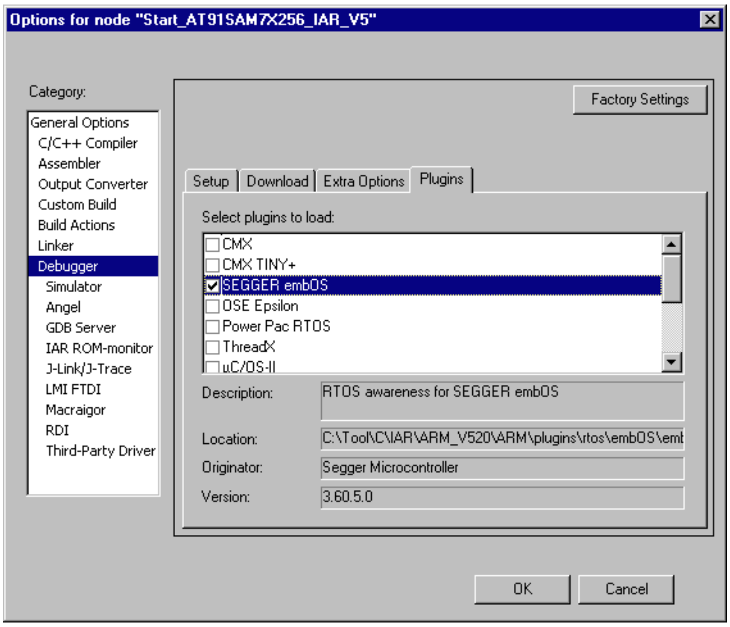
C:\Tool\C\IAR\ARM_V520\ARM\plugins\rtos\embOS
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Tool\C\IAR\ARM_V520\ARM\plugins\rtos\embOS
Go
Folders
ARM_V520
ARM
bin
config
doc
drivers
examples
inc
lib
plugins
rtos
embOS
OSE
PowerPac
2 object(s) (Disk free space: 149 GB)
388 KB
My Computer

Name	Size	Type
embOSPlugin.dll	388 KB	Application Extension
embOSPlugin.ewplugin	1 KB	EWPLUGIN File

If not already delivered with the IAR Embedded Workbench IDE, create a directory `embOS` below the CPU specific `plugin\rtos\` folder and copy the files from the embOS folder which comes with the plug-in into that folder in your IAR installation directory. Then restart the IAR Embedded Workbench IDE.

11.3 Configuration

By default, the embOS C-Spy plug-in is not loaded during debugging. For each project configuration you have to explicitly enable the plug-in in the debugger section of the project options:



Select plugins to load:

□ CMX
□ CMX TINY+
☑ SEGGER embOS
□ OSE Epsilon
□ Power Pac RTOS
□ ThreadX
□ uC/OS-II

Description: RTOS awareness for SEGGER embOS

Location: C:\Tool\C\IAR\ARM_V520\ARM\plugins\rtos\embOS\emb

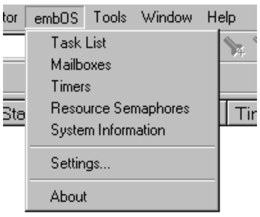
Originator: Segger Microcontroller

Version: 3.60.5.0

The embOS C-Spy plug-in is now available in debugging sessions and may be accessed from the main menu.

11.4 Using the embOS C-Spy plug-in

During your debugging session, the embOS C-Spy plug-in is accessible from the IAR Embedded Workbench IDE **main** menu. Note that if you are not running a debugging session, there is no **embOS menu** item available.

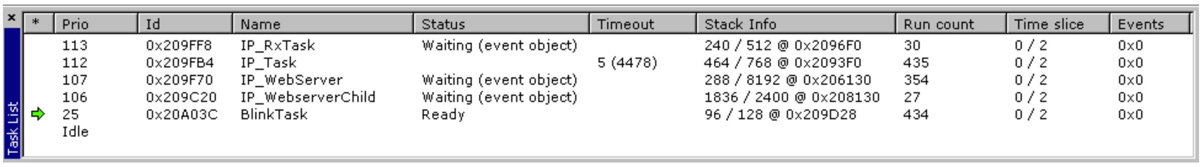


From the menu you may activate the individual windows that provide embOS related information. The sections below describe these individual windows. The amount of information available depends on the embOS build used during debugging. If a certain part is not available, the respective menu item is either greyed out or the window column shows a **N/A**.

11.4.1 Tasks

The **Task List** window lists all embOS tasks. It retrieves its information directly from the embOS task list. The green arrow points to the running task, which is the task currently executing. If no task is executing, the CPU is executing the Idle-loop. In this case, the green arrow is in the bottom row of the window, labeled “Idle”.

The bottom row of the task list window is always labeled “Idle”. It does not actually represent a task, but the Idle loop, which is executed if no task is ready for execution.



*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	113	0x209FF8	IP_RxTask	Waiting (event object)		240 / 512 @ 0x2096F0	30	0 / 2	0x0
	112	0x209FB4	IP_Task		5 (4478)	464 / 768 @ 0x2093F0	435	0 / 2	0x0
	107	0x209F70	IP_WebServer	Waiting (event object)		288 / 8192 @ 0x206130	354	0 / 2	0x0
	106	0x209C20	IP_WebserverChild	Waiting (event object)		1836 / 2400 @ 0x208130	27	0 / 2	0x0
➔	25	0x20A03C	BlinkTask	Ready		96 / 128 @ 0x209D28	434	0 / 2	0x0
	Idle								

The individual columns are described below:

Column	Description
*****	A green arrow points to the running task.
Prio	Priority of the task.
Id	The task control block address that uniquely identifies a task.
Name	If available, the task name is shown here.
Status	The task status as a short text.
Timeout	If a task is delayed, this column shows the time remaining until the delay expires and in parenthesis the time of expiration.
Stack Info	If available, this column shows the amount of used stack space, and the available stack space, as well as the value of the current stack bottom pointer.
Run count	The number of task activations.
Time slice	If round robin scheduling is available, this column shows the number of remaining time slices and the number of time slice reloads.
Events	The event mask of a task.

Table 11.1: Task list window items

11.4.1.1 Task sensitivity

The **Source Code** window, the **Disassembly** window, the **Register** window, and the **Call Stack** window of the C-Spy debugger are task sensitive since version 3.62 of the embOS C-Spy plug-in. This means that they show the position in the code, the general-purpose registers and the call stack of the selected task. By default, the selected task is always the running task, which is the normal behavior of a debugger that the user expects.

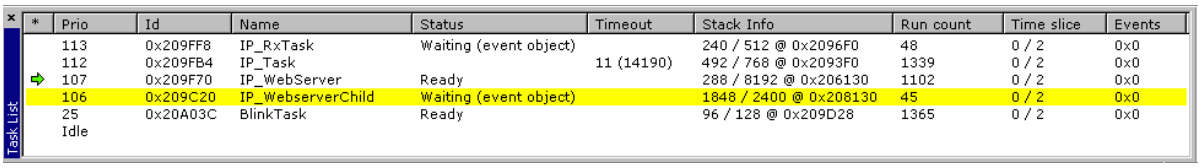
You can examine a particular thread by double-clicking on the corresponding row in the window. The selected task will be underlayed in yellow. The C-Spy Debugger rebuilds the call stack and the preserved general-purpose registers of a suspended task. Refer to *State of suspended tasks* on page 87 for detailed information about which information are available for the different task states.

Every time the CPU is started or when the Idle-row of the task window is double clicked, the selected task is switched back to this default.

11.4.1.2 State of suspended tasks

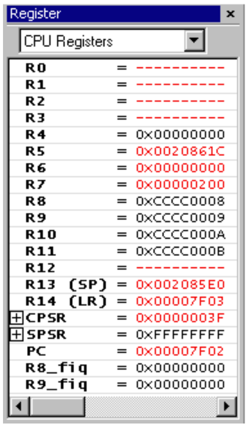
Blocked tasks (suspended by cooperative task switch)

Tasks which have given up execution voluntarily by calling a blocking function, such as `OS_Delay()` or `OS_Wait_...()`. In this case, there was no need for the OS to save the scratch registers (in case of ARM R0-R3, R12).
The **Register** window will show “----------” for the content of these registers.



Task List

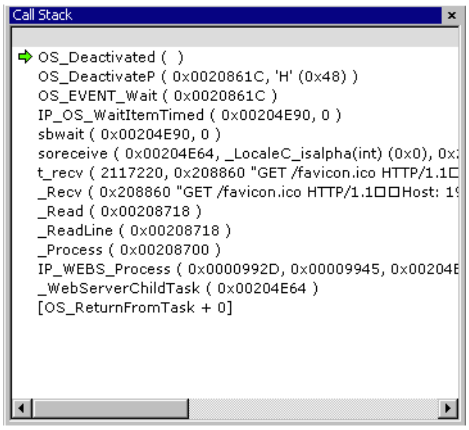
*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	113	0x209FF8	IP_RxTask	Waiting (event object)		240 / 512 @ 0x2096F0	48	0 / 2	0x0
	112	0x209FB4	IP_Task		11 (14190)	492 / 768 @ 0x2093F0	1339	0 / 2	0x0
➪	107	0x209F70	IP_WebServer	Ready		288 / 8192 @ 0x206130	1102	0 / 2	0x0
	106	0x209C20	IP_WebserverChild	Waiting (event object)		1848 / 2400 @ 0x208130	45	0 / 2	0x0
	25	0x20A03C	BlinkTask	Ready		96 / 128 @ 0x209D28	1365	0 / 2	0x0
	Idle								



Register

CPU Registers

R0	=	----------
R1	=	----------
R2	=	----------
R3	=	----------
R4	=	0x00000000
R5	=	0x0020861C
R6	=	0x00000000
R7	=	0x00000200
R8	=	0xCCCC0008
R9	=	0xCCCC0009
R10	=	0xCCCC000A
R11	=	0xCCCC000B
R12	=	----------
R13 (SP)	=	0x002085E0
R14 (LR)	=	0x00007F03
⊞CPSR	=	0x0000003F
⊞SPSR	=	0xFFFFFFFF
PC	=	0x00007F02
R8_fiq	=	0x00000000
R9_fiq	=	0x00000000

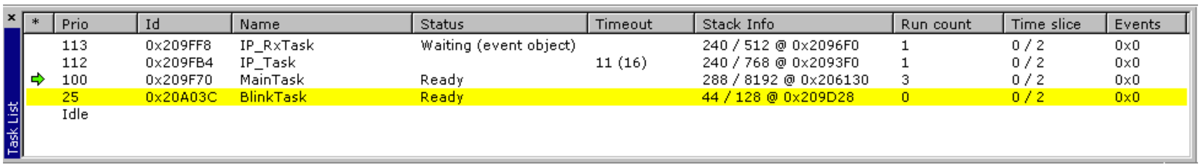


Call Stack

```
➪ OS_Deactivated (  )
  OS_DeactivateP ( 0x0020861C, 'H' (0x48) )
  OS_EVENT_Wait ( 0x0020861C )
  IP_OS_WaitItemTimed ( 0x00204E90, 0 )
  sbwait ( 0x00204E90, 0 )
  soreceive ( 0x00204E64, _LocaleC_isalpha(int) (0x0), 0x
  t_recv ( 2117220, 0x208860 "GET /favicon.ico HTTP/1.1□
  _Recv ( 0x208860 "GET /favicon.ico HTTP/1.1□□Host: 1
  _Read ( 0x00208718 )
  _ReadLine ( 0x00208718 )
  _Process ( 0x00208700 )
  IP_WEBS_Process ( 0x0000992D, 0x00009945, 0x00204E
  _WebServerChildTask ( 0x00204E64 )
  [OS_ReturnFromTask + 0]
```

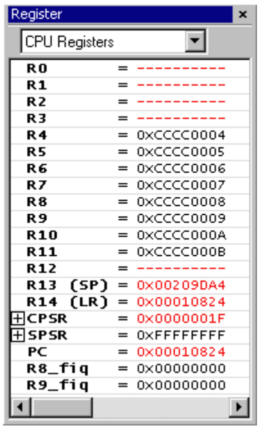
Tasks waiting for first activation

These basically fall into the same category as blocked tasks, the call stack and registers look similar to the following screenshots. Similarly, temporary registers are unknown. The **Call Stack** shows a single entry **OS_StartTask**. **Run count** is 0.



Task List

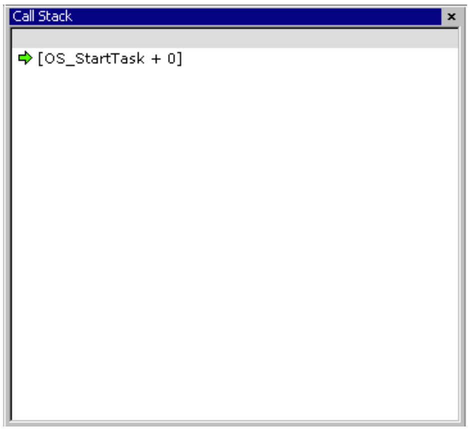
*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	113	0x209FF8	IP_RxTask	Waiting (event object)		240 / 512 @ 0x2096F0	1	0 / 2	0x0
	112	0x209FB4	IP_Task		11 (16)	240 / 768 @ 0x2093F0	1	0 / 2	0x0
➪	100	0x209F70	MainTask	Ready		288 / 8192 @ 0x206130	3	0 / 2	0x0
	25	0x20A03C	BlinkTask	Ready		44 / 128 @ 0x209D28	0	0 / 2	0x0
	Idle								



Register

CPU Registers

R0	=	----------
R1	=	----------
R2	=	----------
R3	=	----------
R4	=	0xCCCC0004
R5	=	0xCCCC0005
R6	=	0xCCCC0006
R7	=	0xCCCC0007
R8	=	0xCCCC0008
R9	=	0xCCCC0009
R10	=	0xCCCC000A
R11	=	0xCCCC000B
R12	=	----------
R13 (SP)	=	0x00209DA4
R14 (LR)	=	0x00010824
⊞CPSR	=	0x0000001F
⊞SPSR	=	0xFFFFFFFF
PC	=	0x00010824
R8_fiq	=	0x00000000
R9_fiq	=	0x00000000

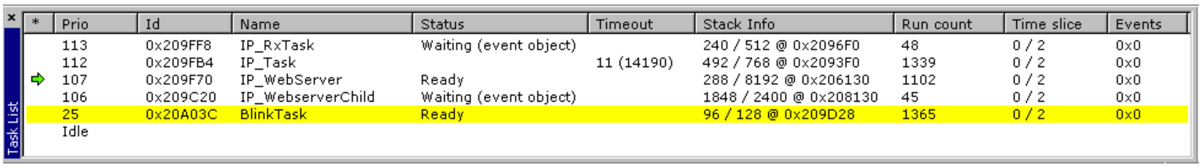


Call Stack

```
➪ [OS_StartTask + 0]
```

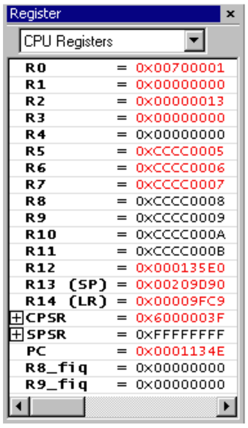
Interrupted tasks

Tasks which have been interrupted and preempted, typically by a task with higher priority becoming ready. In this case, the OS saved all registers, including the scratch registers (in case of ARM R0-R3, R12). The **Register** window shows the values of all registers, including the scratch registers.



Task List

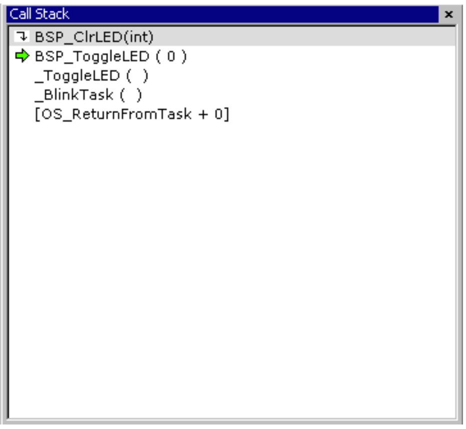
*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	113	0x209FF8	IP_RxTask	Waiting (event object)		240 / 512 @ 0x2096F0	48	0 / 2	0x0
	112	0x209FB4	IP_Task		11 (14190)	492 / 768 @ 0x2093F0	1339	0 / 2	0x0
➡	107	0x209F70	IP_WebServer	Ready		288 / 8192 @ 0x206130	1102	0 / 2	0x0
	106	0x209C20	IP_WebserverChild	Waiting (event object)		1848 / 2400 @ 0x208130	45	0 / 2	0x0
	25	0x20A03C	BlinkTask	Ready		96 / 128 @ 0x209D28	1365	0 / 2	0x0
	Idle								



Register

CPU Registers

R0	=	0x00700001
R1	=	0x00000000
R2	=	0x00000013
R3	=	0x00000000
R4	=	0x00000000
R5	=	0xCCCC0005
R6	=	0xCCCC0006
R7	=	0xCCCC0007
R8	=	0xCCCC0008
R9	=	0xCCCC0009
R10	=	0xCCCC000A
R11	=	0xCCCC000B
R12	=	0x000135E0
R13 (SP)	=	0x00209D90
R14 (LR)	=	0x00009FC9
⊞CPSR	=	0x6000003F
⊞SPSR	=	0xFFFFFFFF
PC	=	0x0001134E
R8_fiq	=	0x00000000
R9_fiq	=	0x00000000

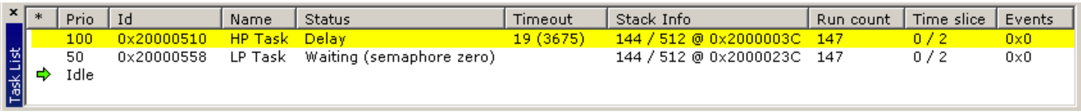


Call Stack

BSP_ClrLED(int)
➡ BSP_ToggleLED (0)
_ToggleLED ( )
_BlinkTask ( )
[OS_ReturnFromTask + 0]

11.4.1.3 Call stack with embOS libraries

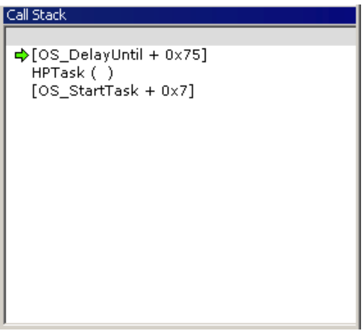
All embOS libraries are built with full optimization. Therefore it may happen that not all function calls are shown in the call stack in detail. The additional embOS library *dpl.a is built with low optimization. It may be used for application development instead of the Debug and Profiling library.
This gives the ability to see the complete detailed call stack.



Task List

*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	100	0x20000510	HP Task	Delay	19 (3675)	144 / 512 @ 0x2000003C	147	0 / 2	0x0
	50	0x20000558	LP Task	Waiting (semaphore zero)		144 / 512 @ 0x2000023C	147	0 / 2	0x0
➡	Idle								

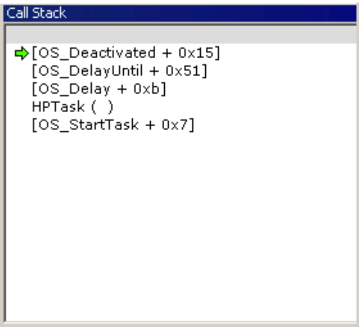
Call stack with DP library



Call Stack

➡[OS_DelayUntil + 0x75]
HPTask ( )
[OS_StartTask + 0x7]

Call stack with DPL library

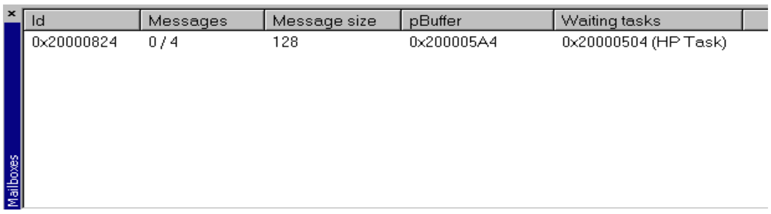


Call Stack

➡[OS_Deactivated + 0x15]
[OS_DelayUntil + 0x51]
[OS_Delay + 0xb]
HPTask ( )
[OS_StartTask + 0x7]

11.4.2 Mailboxes

A mailbox is a buffer that is managed by the real-time operating system. The buffer behaves like a normal buffer; you can put something (called a message) in and retrieve it later. This window shows the mailboxes and provides information about the number of messages, waiting tasks etc.



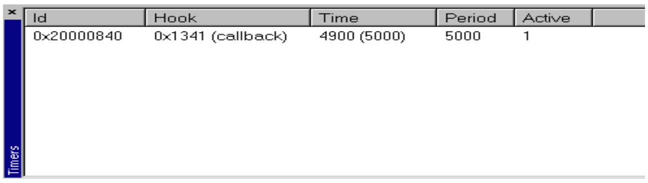
Id	Messages	Message size	pBuffer	Waiting tasks
0x20000824	0 / 4	128	0x200005A4	0x20000504 (HP Task)

Column	Description
Id	The mailbox address.
Messages	The number of messages in a mailbox and the maximum number of messages as mailbox can hold.
Message size	The size of an individual message in bytes.
pBuffer	The message buffer address.
Waiting tasks	The list of tasks that are waiting for a mailbox, that is their address and name.

Table 11.2: Mailboxes window items

11.4.3 Timers

A software timer is an object that calls a user-specified routine after a specified delay. This window provides information about active software timers.



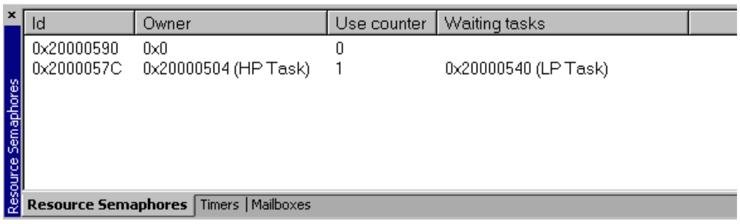
Id	Hook	Time	Period	Active
0x20000840	0x1341 (callback)	4900 (5000)	5000	1

Column	Description
Id	The timer’s address.
Hook	The function (address and name) that is called after the timeout.
Time	The time delay and the point in time, when the timer finishes waiting.
Period	The time period the timer runs.
Active	Shows whether the timer is active or not.

Table 11.3: Timers window items

11.4.4 Resource semaphores

Resource semaphores are used to manage resources by avoiding conflicts caused by simultaneous use of a resource. This window provides information about available resources.



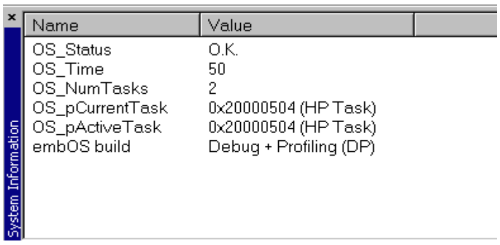
Id	Owner	Use counter	Waiting tasks
0x20000590	0x0	0	
0x2000057C	0x20000504 (HP Task)	1	0x20000540 (LP Task)

Column	Description
Id	The resource semaphore address.
Owner	The address and name of the owner task.
Use counter	Counts the number of semaphore uses.
Waiting tasks	Lists the tasks (address and name) that are waiting at the semaphore.

Table 11.4: Resource Semaphores window items

11.4.5 System information

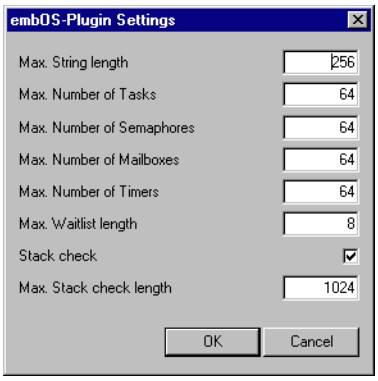
A running embOS contains a number of system variables that are available for inspection. This window lists the most important ones.



Name	Value
OS_Status	O.K.
OS_Time	50
OS_NumTasks	2
OS_pCurrentTask	0x20000504 (HP Task)
OS_pActiveTask	0x20000504 (HP Task)
embOS build	Debug + Profiling (DP)

11.4.6 Settings

To be safe, the embOS C-Spy plug-in imposes certain limits on the amount of information retrieved from the target, to avoid endless requests in case of false values in the target memory. This dialog box allows you to tweak these limits in a certain range, for example if your task names are no longer than 32 characters you may set the **Maximum string length** to 32, or if they are longer than the default you may increase that value.



Max. String length 256
Max. Number of Tasks 64
Max. Number of Semaphores 64
Max. Number of Mailboxes 64
Max. Number of Timers 64
Max. Waitlist length 8
Stack check ☑
Max. Stack check length 1024

After changing settings and clicking the **OK** button, your changes are applied immediately and should become noticeable after the next window update, for example when hitting the next breakpoint. However, the settings are restored to their default values on plug-in reload.

11.4.7 About

Finally, the **About** dialog box contains the embOS C-Spy plug-in version number and the date of compilation.

Chapter 12

Files shipped with embOS

List of 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\BoardSupport\		Sample workspaces and project files for IAR Embedded Workbench, contained in manufacturer specific sub folders.
Start\Inc	RTOS.h BSP.h	Include file for embOS, to be included in every C-file using embOS functions.
Start\Lib	os??_*.a	embOS libraries for IAR compiler V6.x
Start\BoardSupport\..\Setup	OS_Error.c	embOS runtime error handler used in stack check or debug builds.
Start\BoardSupport\...\Setup\	*.*	CPU specific hardware routines for various CPUs.

Table 12.1: Files shipped with embOS

Any additional files shipped serve as example.

Index