embOS

Real-Time Operating System

CPU & Compiler specifics for Texas Instruments MSP430 CPUs using TI Code Composer for MSP430

Document: UM01066
Software Version: 5.18.0.0
Revision: 0
Date: October 18, 2022

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	support@segger.com*
Internet:	*www.segger.com*

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: October 18, 2022

Software	Revision	Date	By	Description
5.18.0.0	0	221018	TS	New software version.
4.26	0	161018	RH	Chapters "Libraries" and "CPU and compiler specifics" introduced.
4.00	0	140617	SC	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).
- The C programming language.
- The target processor.
- DOS command line.

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

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

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.
`Sample comment`	Comments 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 of contents

Chapter 1

Using embOS

1.1 Installation

This chapter describes how to start with embOS. You should follow these steps to become familiar with embOS.

embOS is shipped as a zip-file in electronic form.

To install it, proceed as follows:

Extract the zip-file to any folder of your choice, preserving the directory structure of this file. Keep all files in their respective sub directories. Make sure the files are not read only after copying.

> **Note**
>
> The BSP projects at `/Start/BoardSupport/<DeviceManufacturer>/<Device>` assume that the `/Start/Lib` and `Start//Inc` folders are located relative to the BSP folder. If you copy a BSP folder to another location, you will need to adjust these paths in the project.

Assuming that you are using an IDE to develop your application, no further installation steps are required. You will find many prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section *First Steps* on page 10.

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

If you do not or do not want to work with the IDE, you should: Copy either all or only the library-file that you need to your work-directory. The advantage is that when switching to an updated version of embOS later in a project, you do not affect older projects that use embOS, too. embOS does in no way rely on an IDE, it may be used without the IDE using batch files or a make utility without any problem.

1.2 First Steps

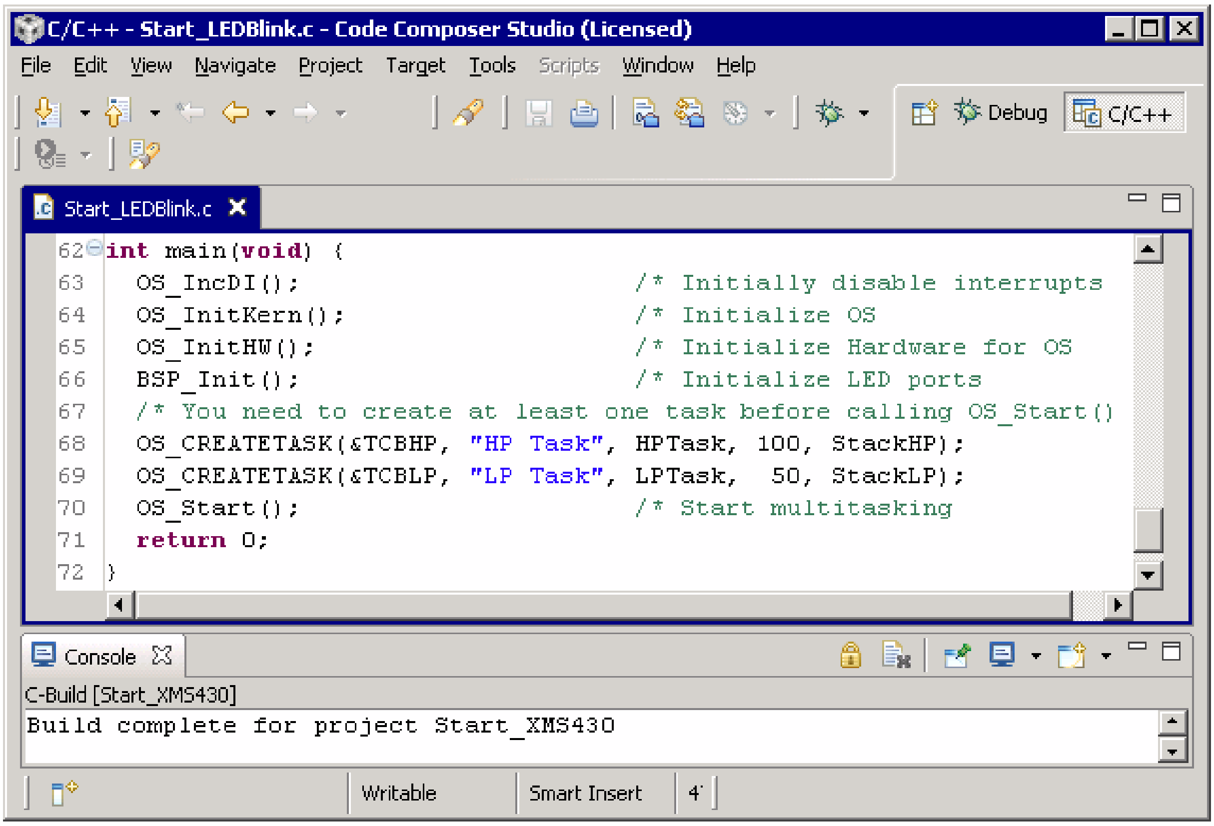
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder `Start`. It is a good idea to use one of them as a starting point for all of your applications. The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

To start with, you may use any project from `BoardSupport` subfolder.

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 one sample workspace/project in `Start\BoardSupport\<DeviceManufacturer>\<CPU>` with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

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



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 OS_StartLEDBlink.c

The following is a printout of the example application `OS_StartLEDBlink.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 GmbH                    *
*                        The Embedded Experts                        *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------
File    : OS_StartLEDBlink.c
Purpose : embOS sample program running two simple tasks, each toggling
          a LED of the target hardware (as configured in BSP.c).
*/

#include "RTOS.h"
#include "BSP.h"

static OS_STACKPTR int StackHP[128], StackLP[128];  // Task stacks
static OS_TASK         TCBHP, TCBLP;                // Task control blocks

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

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

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_Init();    // Initialize embOS
  OS_InitHW();  // Initialize required hardware
  BSP_Init();   // Initialize LED ports
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();   // Start embOS
  return 0;
}

/*************************** End of file ****************************/
```

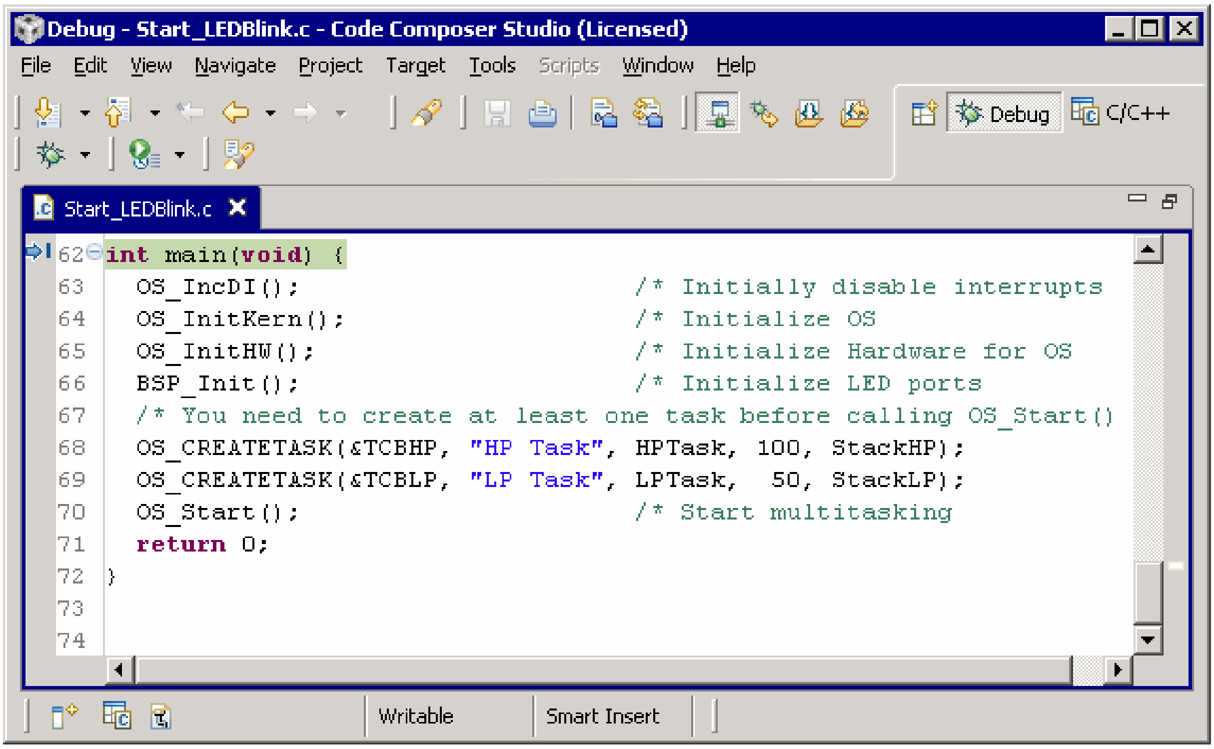

1.4 Stepping through the sample application

When starting the debugger, you will see the `main()` function (see example screenshot below). The `main()` function appears as long as project option `Run to main` is selected, which it is enabled by default. Now you can step through the program.

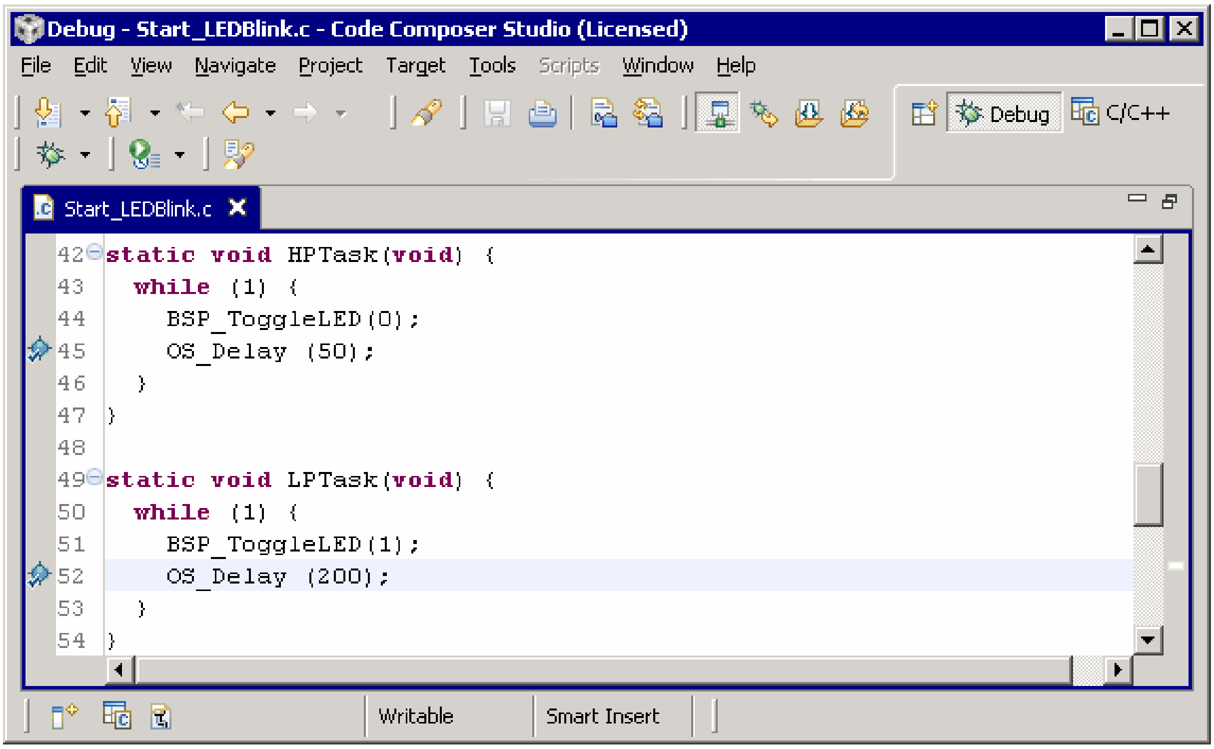
`OS_Init()` is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`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 system 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.

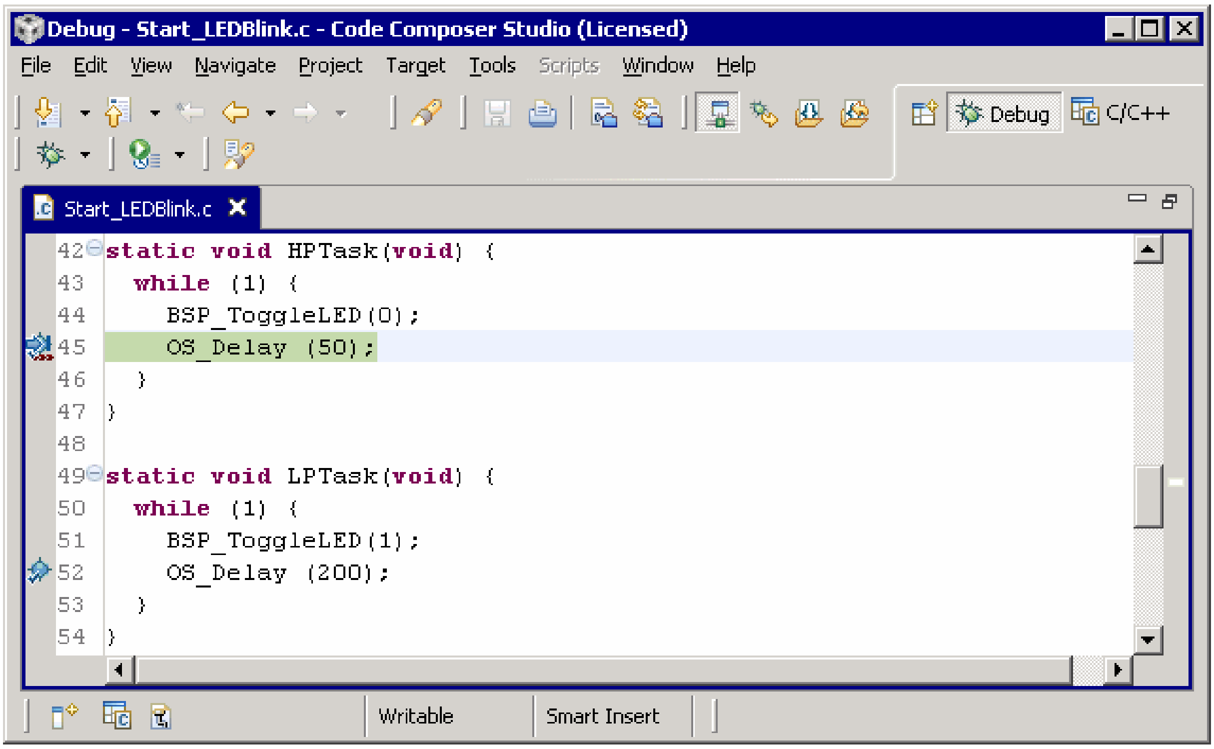


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

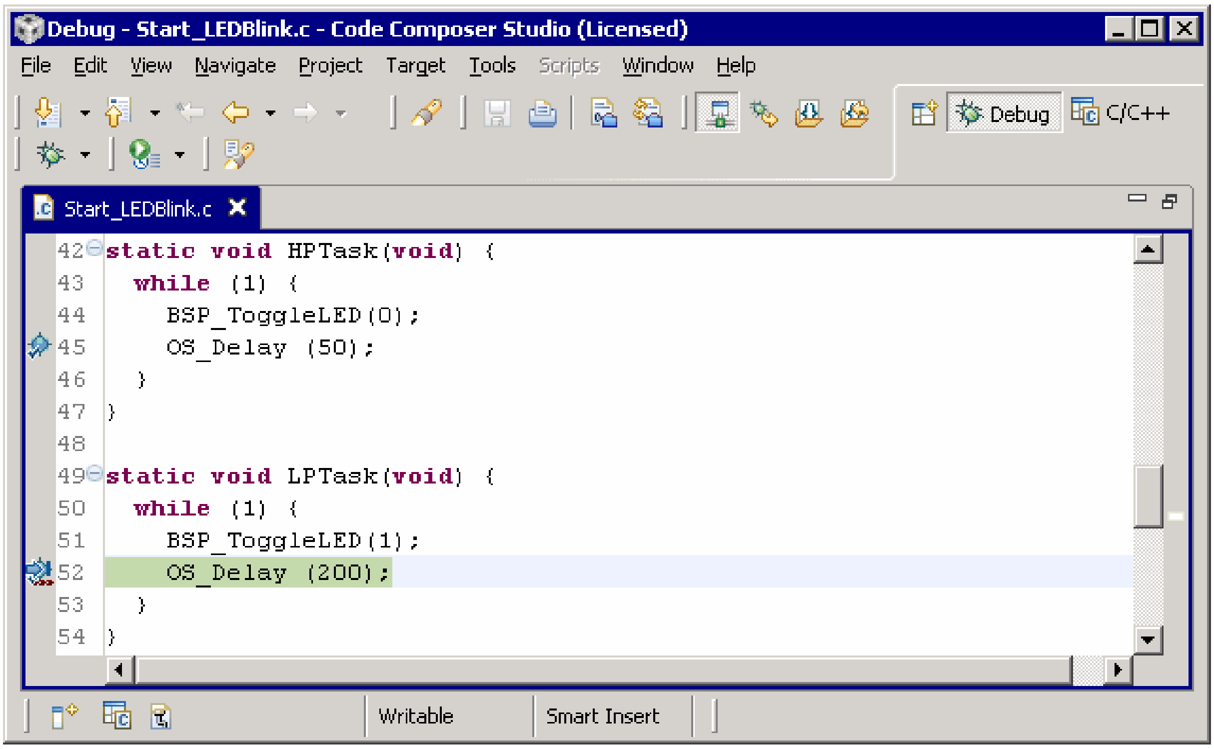


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

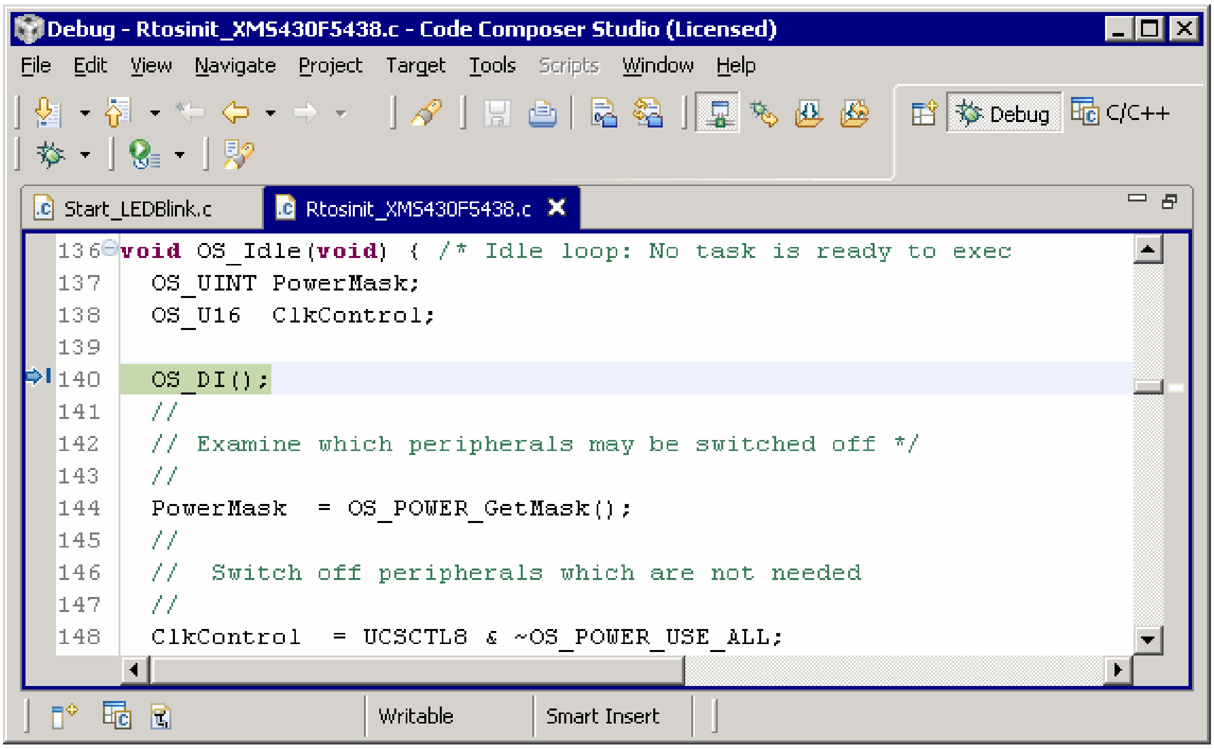


If you continue stepping, you will arrive at the task that has lower priority:



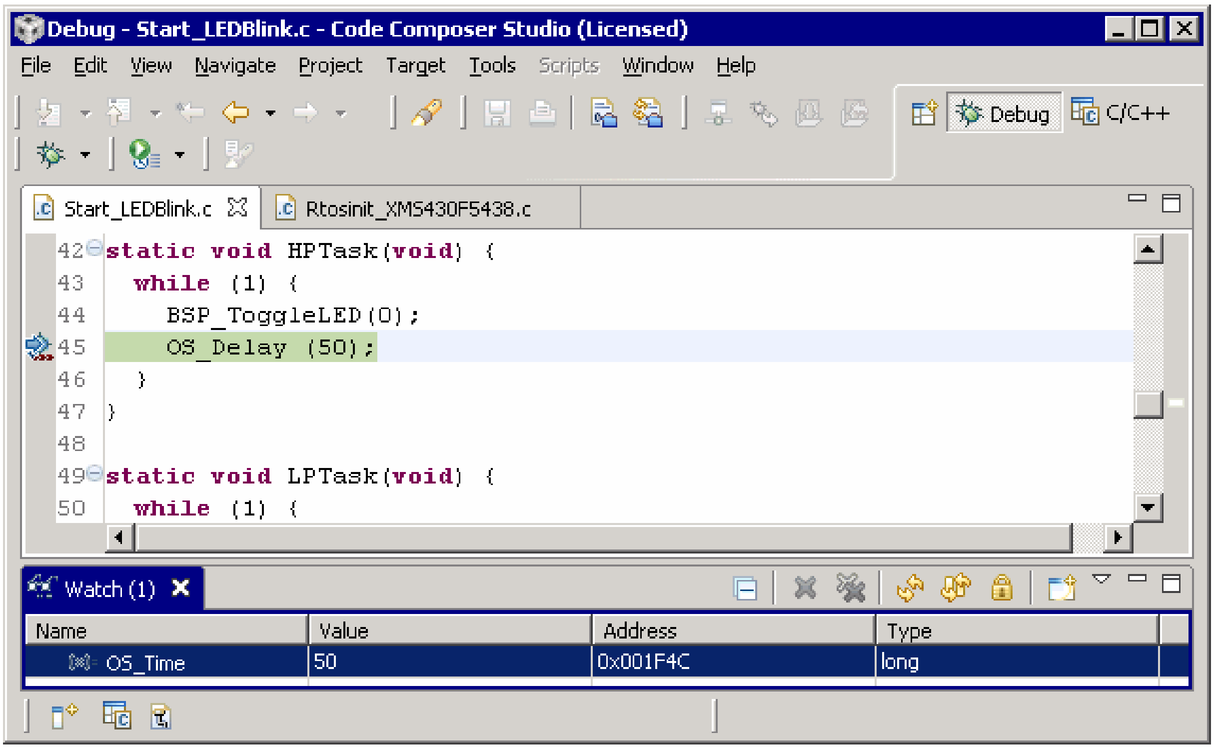
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

You will arrive there when you step into the `OS_TASK_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit.c`. You may also set a breakpoint there before stepping over the delay in `LPTask()`.



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_Global.Time`, shown in the Watch window, `HPTask()` continues operation after expiration of the delay.



Chapter 2

Build your own application

2.1 Introduction

This chapter provides all information to set up your own embOS project. 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 chapter *First Steps* on page 10 and modify the project to fit your needs. Using an embOS start 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

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 the directory `.\Start\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 `.\Start\BoardSupport\<Manufacturer>\<MCU>` subfolder. It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt but can also initialize or set up the interrupt controller, clocks and PLLs, the memory protection unit and its translation table, caches and so on.
- **`OS_Error.c`** from one target specific subfolder `.\Start\BoardSupport\<Manufacturer>\<MCU>`. The error handler is used only if a debug library is used in your project.
- One **embOS library** from the subfolder `.\Start\Lib`.
- Additional CPU and compiler specific files 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. Your `main()` function has to initialize embOS by calling `OS_Init()` and `OS_InitHW()` prior to any other embOS functions that are called.

2.3 Change library mode

For your application you might want to choose another library. For debugging and program development you should always 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 project configuration.
- To add a library, you may add the library to the existing Lib group. Exclude all other libraries from your build, delete unused libraries 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 CPUs. Manufacturer- and CPU-specific sample start workspaces and projects are located in the subfolders of the `.\Start\BoardSupport` directory. To select a CPU which is already supported, just select the appropriate workspace from a CPU-specific folder.

If your CPU is currently not supported, examine all `RTOSInit.c` files in the CPU-specific subfolders and select one which almost fits your CPU. You may have to modify `OS_InitHW()`, the interrupt service routines for the embOS system tick timer and the low level initialization.

Chapter 3

Libraries

3.1 Naming conventions for prebuilt libraries

embOS is shipped with different pre-built libraries with different combinations of features.

The libraries are named as follows:

`<CPU><DataModel>_<LibMode>.lib`

Parameter	Meaning	Values
CPU	Specifies the CPU variant.	430: MSP430 430x: MSP430x, extended addresses
DataModel	Selected data model, only for MSP430x CPUs.	: small data model l: large data model
LibMode	Specifies the library mode.	XR: Extreme Release R: Release S: Stack check SP: Stack check + profiling D: Debug DP: Debug + profiling DT: Debug + profiling + trace

Example

`430xl_SP.lib` is the embOS library for an MSP430X CPU with large memory model and stack check and profiling functionality.

Chapter 4

CPU and compiler specifics

4.1 Interrupt and thread safety

Using embOS with specific calls to standard library functions (e.g. heap management functions) may require thread-safe system libraries if these functions are called from several tasks or interrupts. TI Code Composer's system library provides functionality to implement a locking mechanism making the system library functions thread-safe.

The Setup directory in each embOS BSP contains the file `OS_ThreadSafe.c`. By default it disables and restores embOS interrupts to ensure thread safety in tasks, embOS interrupts, `OS_Idle()` and software timers. Zero latency interrupts are not disabled and therefore unprotected. If you need to call e.g. malloc() also from within a zero latency interrupt additional handling needs to be added. If you don't call such functions from within embOS interrupts, `OS_Idle()` or software timers, you can instead use thread safety for tasks only. This reduces the interrupt latency because a mutex is used instead of disabling embOS interrupts.

You can choose the safety variant with the macro `OS_INTERRUPT_SAFE`.

- When defined to 1 thread safety is guaranteed in tasks, embOS interrupts, `OS_Idle()` and software timers.
- When defined to 0 thread safety is guaranteed only in tasks. In this case you must not call e.g. heap functions from within an ISR, `OS_Idle()` or embOS software timers.

Alternatively, embOS delivers its own thread-safe functions for heap management. These are described in the embOS generic manual.

Chapter 5

Stacks

5.1 Task stack

Each task uses its individual stack. The stack pointer 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 MSP430, this minimum stack size is about 24 bytes and for MSP430X using large data model it is about 46 bytes. As MSP430(X) devices do not support an own interrupt stack, please note, that interrupts can also run on task stacks. You may use embOSView together with an embOS stack check library to analyze the total amount of task stack used in your application. We recommend at least a minimum task stack size of 128 bytes.

5.2 System stack

The minimum system stack size required by embOS is about 60 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 stack size define in your linker file. We recommend a minimum stack size of 128 bytes.

5.3 Interrupt stack

Since MSP430(X) devices do not provide a separate stack pointer for interrupts, every interrupt occupies additional stack space on the current stack. This may be the system stack, or a task stack of a running task that is interrupted. The additional amount of necessary stack for all interrupts has to be reserved on all task stacks. The current version of embOS for MSP430 does not support extra interrupt stack switching in an interrupt routine. `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()` are supplied for source compatibility to other processors only and have no functionality.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU receives an interrupt request.
- As soon as the interrupts are enabled, the interrupt is accepted.
- The CPU saves PC and flags on the stack.
- The CPU jumps to the address specified in the vector table for the interrupt service routine (ISR).
- ISR: save registers (function prologue)
- ISR: user-defined functionality
- ISR: restore registers (function epilogue)
- ISR: Execute RETI command, restoring PC, Flags and continue interrupted program

For details, please refer to Texas Instruments' user's manual.

6.2 Defining interrupt handlers in "C"

Routines defined with the keyword `__interrupt` automatically save & restore the registers they modify and return with RETI. The interrupt vector number has to be given as additional parameter by a `#pragma` directive prior the interrupt handler function.

For a detailed description on how to define an interrupt routine in "C", refer to the IAR C/C++ Compiler reference guide.

Example

Simple interrupt routine:

```
#prgama vector=12
static __interrupt void IntHandlerTimer(void) {
  IntCnt++;
}
```

Interrupt routine calling embOS functions

```
#prgama vector=12
static __interrupt void IntHandlerTimer(void) {
  OS_INT_Enter();  // Inform embOS that interrupt function is running
  IntCnt++;
  OS_MAILBOX_Put(&MB_Data, &IntCnt);
  OS_INT_Leave();
}
```

`OS_INT_Enter()` has to be the first function called in an interrupt handler using embOS functions, when nestable interrupts are not required. `OS_INT_Leave()` has to be called at the end the interrupt handler then. If interrupts should be nested, use `OS_INT_EnterNestable()` and `OS_INT_LeaveNestable()` instead.

> **Note**
>
> MSP430 devices do not provide a separate stack pointer for interrupts, but use the current stack. For more information, please refer to *Interrupt stack* on page 23.

Chapter 7

Technical data

7.1 Resource Usage

The memory requirements of embOS (RAM and ROM) differs depending on the used features, CPU, compiler, and library model. The following values are measured using embOS library mode `OS_LIBMODE_XR`.

Module	Memory type	Memory requirements
embOS kernel	ROM	~1700 bytes
embOS kernel	RAM	~122 bytes
Task control block	RAM	20 bytes
Software timer	RAM	14 bytes
Task event	RAM	0 bytes
Event object	RAM	10 bytes
Mutex	RAM	14 bytes
Semaphore	RAM	6 bytes
RWLock	RAM	22 bytes
Mailbox	RAM	18 bytes
Queue	RAM	20 bytes
Watchdog	RAM	8 bytes
Fixed Block Size Memory Pool	RAM	24 bytes

