embOS

Real-Time Operating System

CPU & Compiler specifics
for MSP430 using GCC

Document: UM01087
Software Version: 5.18.3.0
Revision: 0
Date: October 14, 2024

A product of SEGGER Microcontroller GmbH

www.segger.com

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 14, 2024

Software	Revision	Date	By	Description
5.18.3.0	0	241014	MC	Initial 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.

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 get started 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, you should extract the zip-file to any folder of your choice while preserving its directory structure (i.e. keep all files in their respective sub directories). Ensure the files are not read-only after extraction. Assuming that you are using an IDE to develop your application, no further installation steps are required.

> **Note**
>
> The projects at `/Start/BoardSupport/<DeviceManufacturer>/<Board>` assume a relative location for the `/Start/Lib` and `/Start/Inc` folders. If you copy a BSP folder to another location, you will need to adjust the include paths of the project accordingly.

At `/Start/BoardSupport/<DeviceManufacturer>/<Board>` you should find several example start projects, which you may adapt to write your application. To do so, follow the instructions of section *First Steps* on page 10.

In order to become familiar with embOS, consider using the example projects (even if you will not use the IDE for application development).

If you do not or do not want to work with an IDE, you may copy either all library files or only the library that is used with your project into your work directory. embOS does in not rely on an IDE, but may be used without an IDE just as well, e.g. using batch files or a make utility.

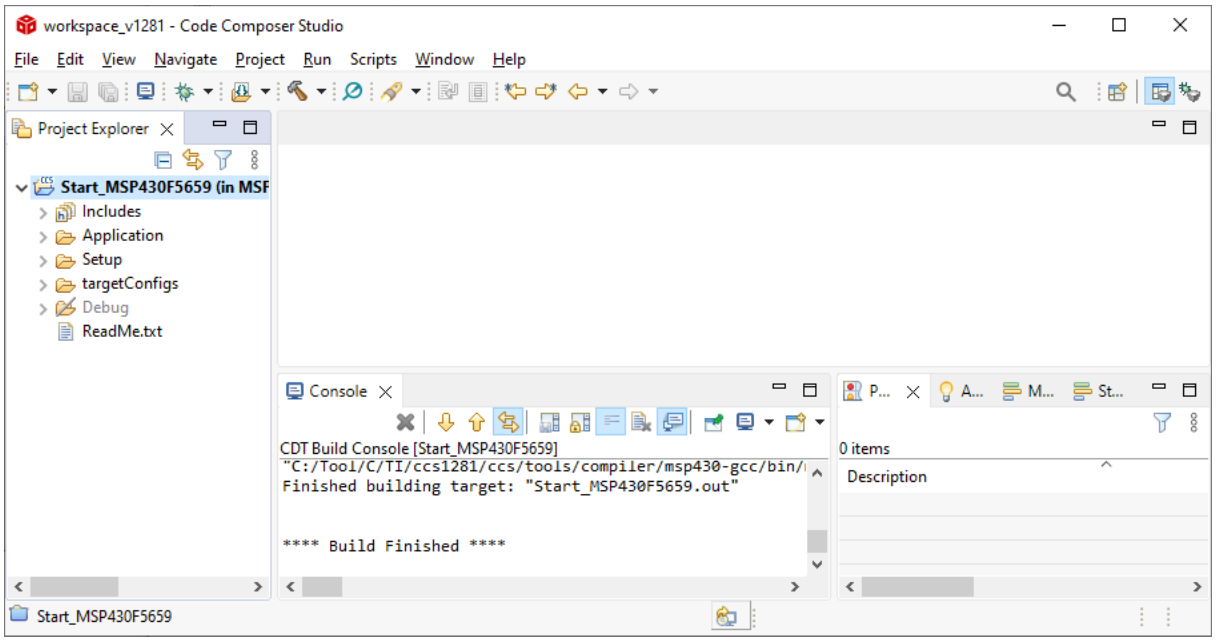
1.2 First Steps

After installation of embOS, you can create your first multitasking application. You received several ready-to-go sample workspaces and projects as well as all required embOS files inside the subfolder `Start`. The subfolder `Start/BoardSupport` contains the workspaces and projects, sorted into manufacturer- and board-specific subfolders. It is a good idea to use one of the projects as a starting point for any application development.

To get your new application running, you should:

- Create a directory for your development.
- Copy the whole `Start` folder from your embOS shipment into the directory.
- Clear the read-only attribute of all files in the copied `Start` folder.
- Open one sample workspace/project in `Start/BoardSupport/<DeviceManufacturer>/<Board>` with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

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



```
CDT Build Console [Start_MSP430F5659]
"C:/Tool/C/TI/ccs1281/ccs/tools/compiler/msp430-gcc/bin/
Finished building target: "Start_MSP430F5659.out"


**** Build Finished ****
```

For additional information, you should open the ReadMe.txt file that is part of every BSP. It describes the different configurations of the project and, if required, gives additional information about specific hardware settings of the supported evaluation board(s).

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 (the actual file shipped with your port of embOS may differ slightly).

What happens is easy to see:

After initialization of embOS, two tasks are created and started. The two tasks get activated and execute until they run into a delay, thereby suspending themselves for the specified time, and eventually 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
          an 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 ****************************/
```

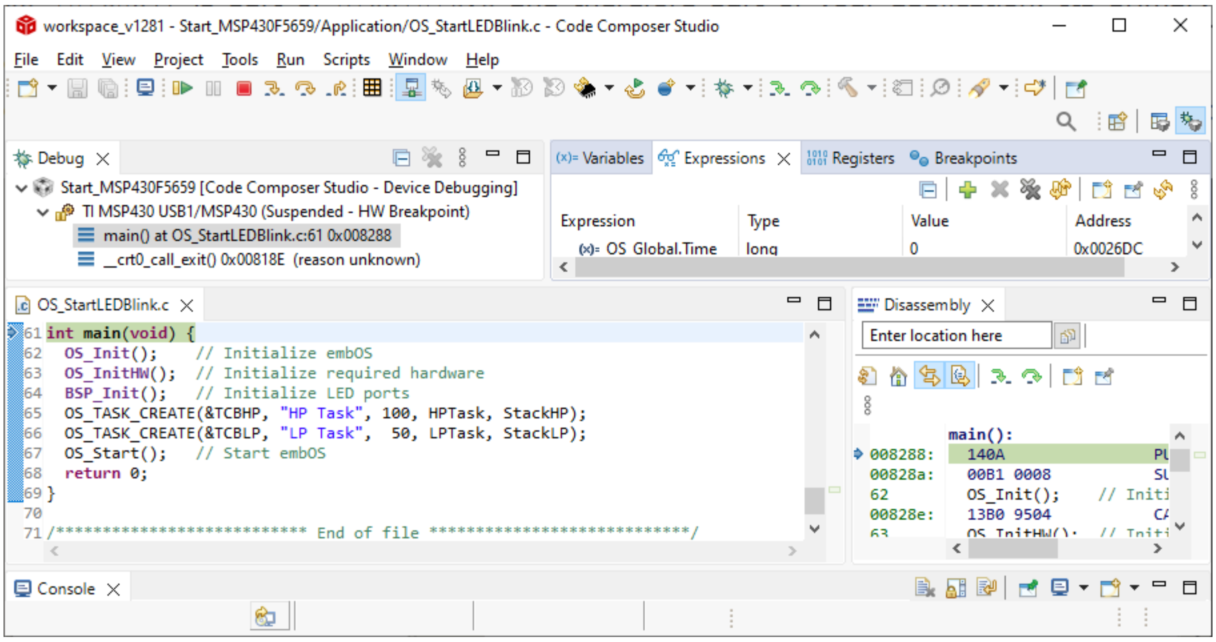
1.4 Stepping through the sample application

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.


workspace_v1281 - Start_MSP430F5659/Application/OS_StartLEDBlink.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
(x)= Variables
Expressions
Registers
Breakpoints
OS_StartLEDBlink.c
Disassembly
Enter location here
Console

Start_MSP430F5659 [Code Composer Studio - Device Debugging]
TI MSP430 USB1/MSP430 (Suspended - HW Breakpoint)
main() at OS_StartLEDBlink.c:61 0x008288
_crt0_call_exit() 0x00818E (reason unknown)

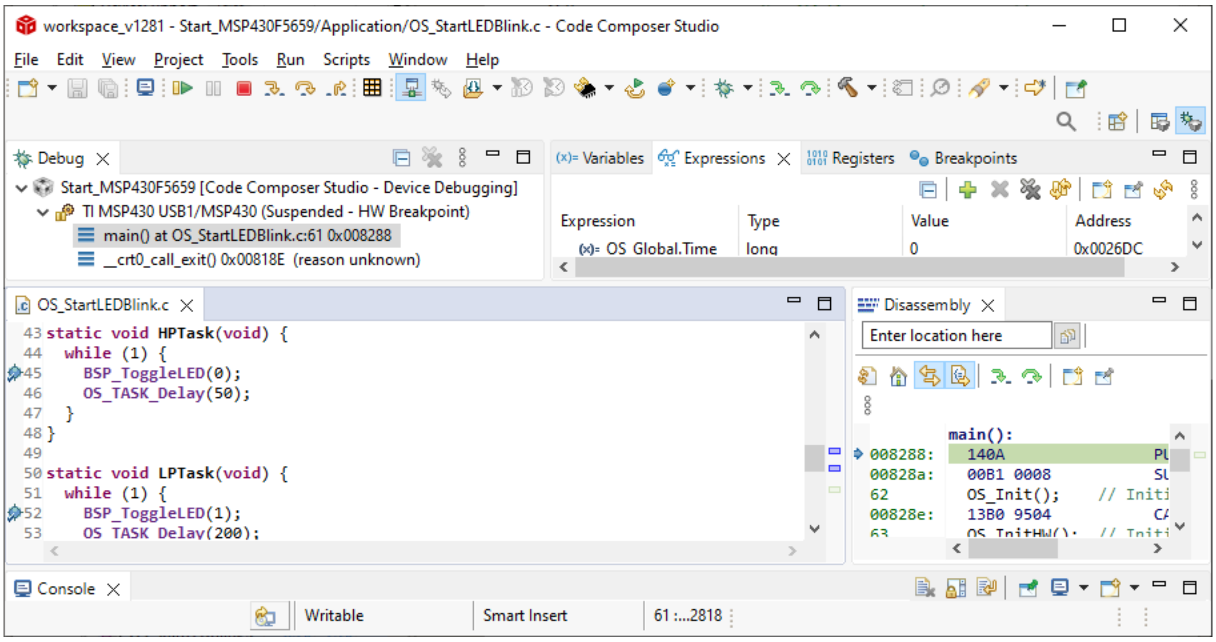
Expression	Type	Value	Address
OS_Global.Time	long	0	0x0026DC

```
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 ****************************/
```

```
          main():
008288:     140A           PU
00828a:     00B1 0008      SU
62          OS_Init();     // Initi
00828e:     13B0 9504      CA
63          OS_InitHW():   // Initi
```

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


workspace_v1281 - Start_MSP430F5659/Application/OS_StartLEDBlink.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
(x)= Variables
Expressions
Registers
Breakpoints
OS_StartLEDBlink.c
Disassembly
Enter location here
Console
Writable
Smart Insert
61 :...2818

Start_MSP430F5659 [Code Composer Studio - Device Debugging]
TI MSP430 USB1/MSP430 (Suspended - HW Breakpoint)
main() at OS_StartLEDBlink.c:61 0x008288
_crt0_call_exit() 0x00818E (reason unknown)

Expression	Type	Value	Address
OS_Global.Time	long	0	0x0026DC

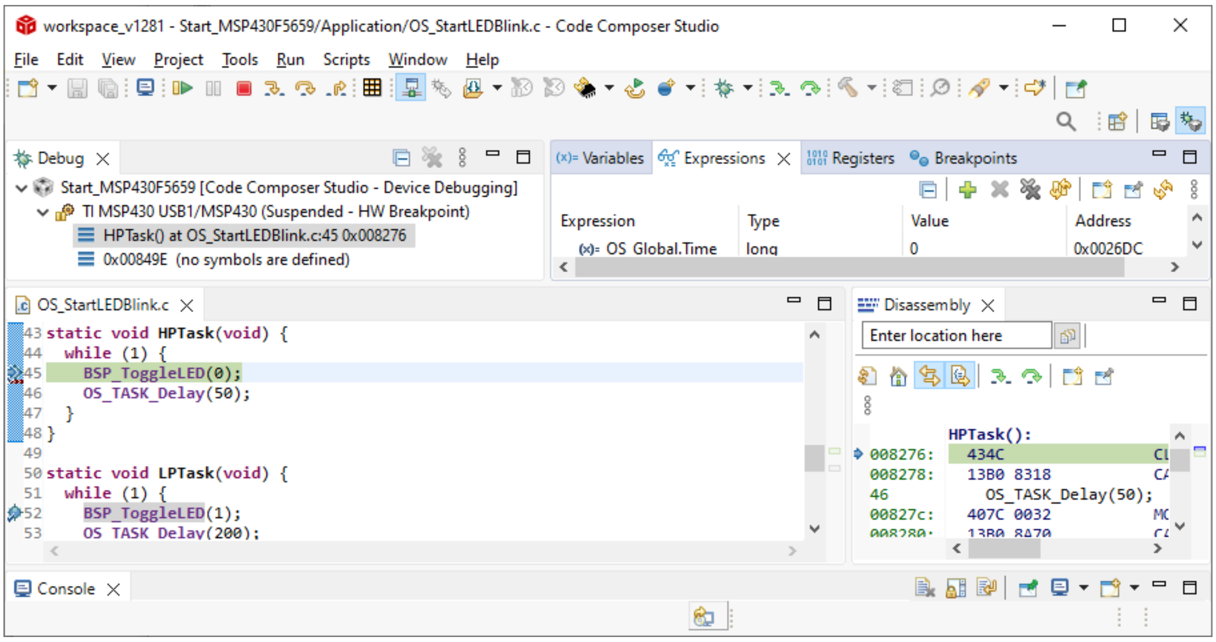
```
static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}

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

```
          main():
008288:     140A           PU
00828a:     00B1 0008      SU
62          OS_Init();     // Initi
00828e:     13B0 9504      CA
63          OS_InitHW():   // Initi
```

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.


workspace_v1281 - Start_MSP430F5659/Application/OS_StartLEDBlink.c - Code Composer Studio

File Edit View Project Tools Run Scripts Window Help
Debug
Variables Expressions Registers Breakpoints
OS_StartLEDBlink.c
Disassembly
Enter location here
Console

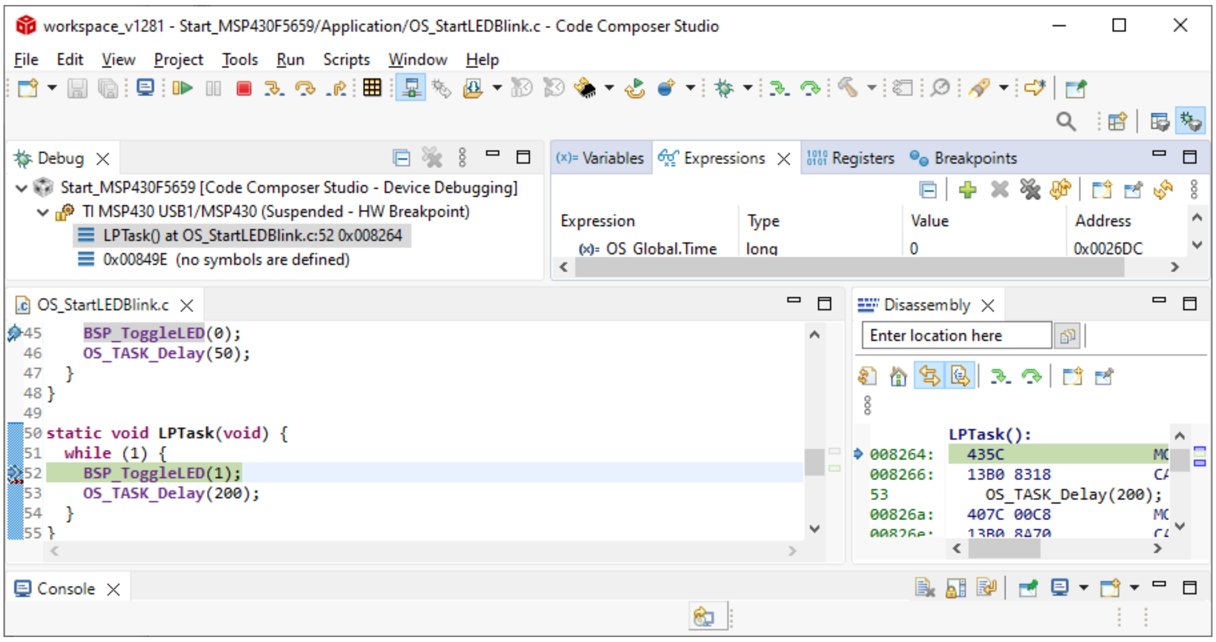
```
Start_MSP430F5659 [Code Composer Studio - Device Debugging]
  TI MSP430 USB1/MSP430 (Suspended - HW Breakpoint)
    HPTask() at OS_StartLEDBlink.c:45 0x008276
    0x00849E  (no symbols are defined)
```

Expression	Type	Value	Address
OS_Global.Time	long	0	0x0026DC

```
43 static void HPTask(void) {
44   while (1) {
45     BSP_ToggleLED(0);
46     OS_TASK_Delay(50);
47   }
48 }
49
50 static void LPTask(void) {
51   while (1) {
52     BSP_ToggleLED(1);
53     OS_TASK_Delay(200);
```

```
          HPTask():
008276:     434C
008278:     13B0 8318
46            OS_TASK_Delay(50);
00827c:     407C 0032
```

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


workspace_v1281 - Start_MSP430F5659/Application/OS_StartLEDBlink.c - Code Composer Studio

File Edit View Project Tools Run Scripts Window Help
Debug
Variables Expressions Registers Breakpoints
OS_StartLEDBlink.c
Disassembly
Enter location here
Console

```
Start_MSP430F5659 [Code Composer Studio - Device Debugging]
  TI MSP430 USB1/MSP430 (Suspended - HW Breakpoint)
    LPTask() at OS_StartLEDBlink.c:52 0x008264
    0x00849E  (no symbols are defined)
```

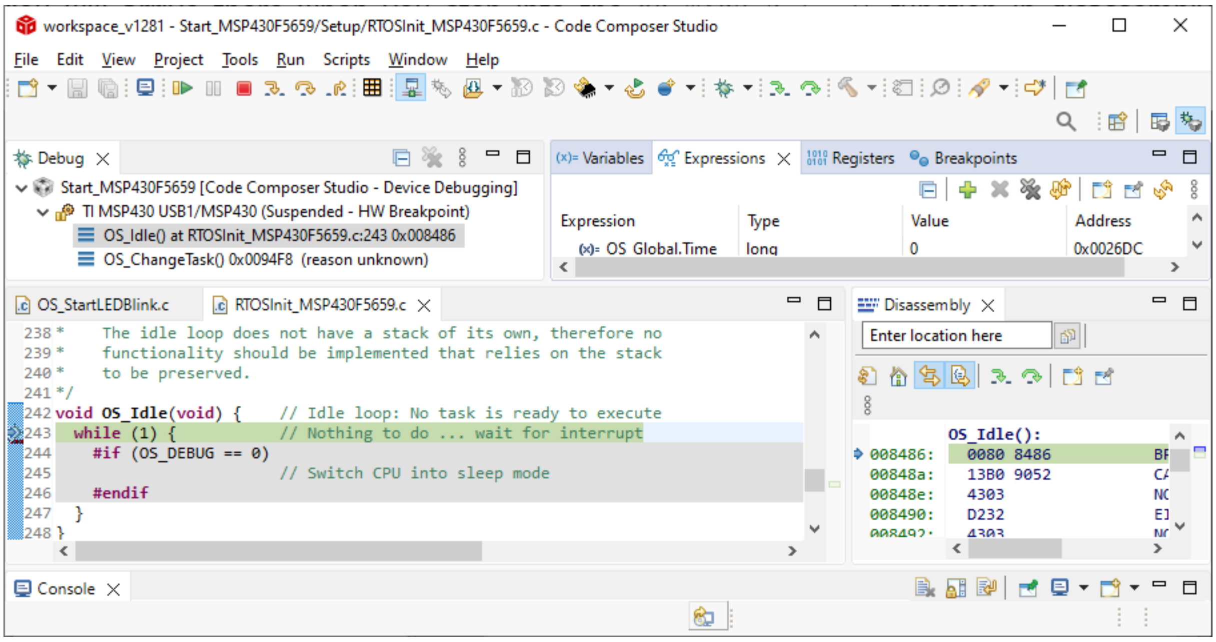
Expression	Type	Value	Address
OS_Global.Time	long	0	0x0026DC

```
45     BSP_ToggleLED(0);
46     OS_TASK_Delay(50);
47   }
48 }
49
50 static void LPTask(void) {
51   while (1) {
52     BSP_ToggleLED(1);
53     OS_TASK_Delay(200);
54   }
55 }
```

```
          LPTask():
008264:     435C
008266:     13B0 8318
53            OS_TASK_Delay(200);
00826a:     407C 00C8
```

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()`.



workspace_v1281 - Start_MSP430F5659/Setup/RTOSInit_MSP430F5659.c - Code Composer Studio

File Edit View Project Tools Run Scripts Window Help
Debug
Variables Expressions Registers Breakpoints
OS_StartLEDBlink.c RTOSInit_MSP430F5659.c
Disassembly
Enter location here
Console

Start_MSP430F5659 [Code Composer Studio - Device Debugging]
TI MSP430 USB1/MSP430 (Suspended - HW Breakpoint)
OS_Idle() at RTOSInit_MSP430F5659.c:243 0x008486
OS_ChangeTask() 0x0094F8 (reason unknown)

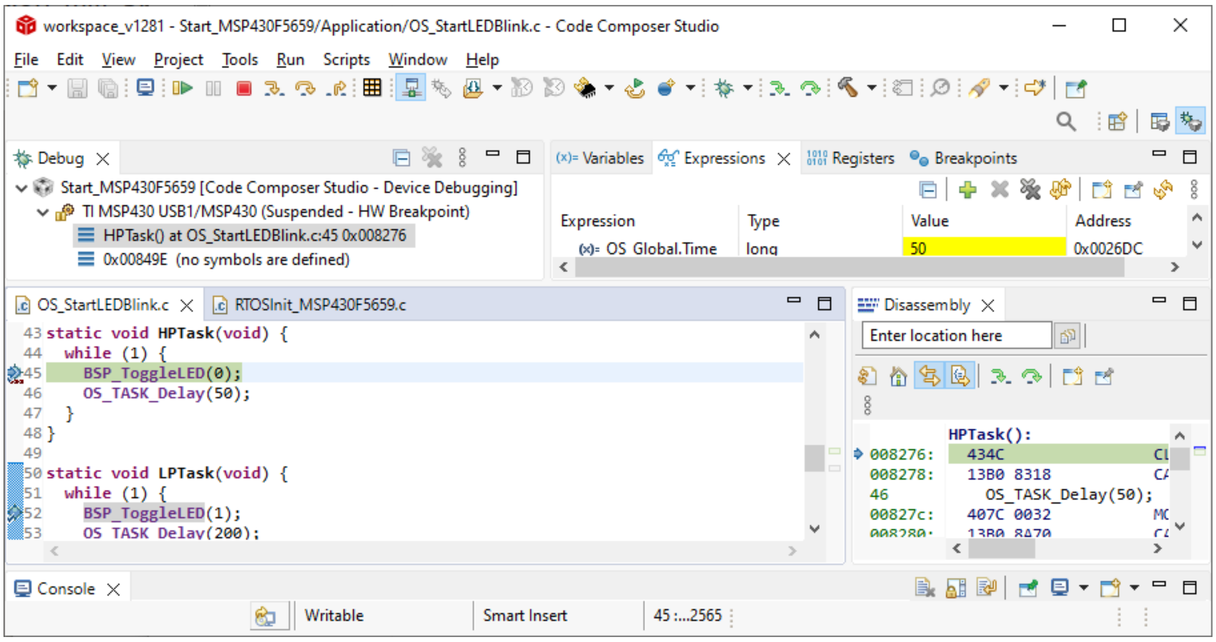
Expression	Type	Value	Address
OS_Global.Time	long	0	0x0026DC

```
 *    The idle loop does not have a stack of its own, therefore no
 *    functionality should be implemented that relies on the stack
 *    to be preserved.
 */
void OS_Idle(void) {     // Idle loop: No task is ready to execute
  while (1) {            // Nothing to do ... wait for interrupt
    #if (OS_DEBUG == 0)
                         // Switch CPU into sleep mode
    #endif
  }
}
```

```
         OS_Idle():
008486:    0080 8486        BF
00848a:    13B0 9052        CA
00848e:    4303             NC
008490:    D232             EI
```

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.



workspace_v1281 - Start_MSP430F5659/Application/OS_StartLEDBlink.c - Code Composer Studio

File Edit View Project Tools Run Scripts Window Help
Debug
Variables Expressions Registers Breakpoints
OS_StartLEDBlink.c RTOSInit_MSP430F5659.c
Disassembly
Enter location here
Console
Writable Smart Insert 45 :...2565

Start_MSP430F5659 [Code Composer Studio - Device Debugging]
TI MSP430 USB1/MSP430 (Suspended - HW Breakpoint)
HPTask() at OS_StartLEDBlink.c:45 0x008276
0x00849E (no symbols are defined)

Expression	Type	Value	Address
OS_Global.Time	long	50	0x0026DC

```
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);
```

```
         HPTask():
008276:    434C             CL
008278:    13B0 8318        CA
46             OS_TASK_Delay(50);
00827c:    407C 0032        MO
```

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 pre-built libraries

embOS is shipped with different pre-built libraries with different combinations of features. Note that not all combinations are available (e.g., there is data model for MSP430, but MSP430x only).

The libraries are named as follows:

`libos<CPU><DataModel>_<LibMode>.a`

Parameter	Meaning	Values
`CPU`	CPU variant	`430` : MSP430 `430x` : MSP430x
`DataModel`	Data model	: small data model `l` : large data model
`LibMode`	embOS library mode	`XR` : Extreme Release `R` : Release `S` : Stack check `SP` : Stack check + profiling `D` : Debug `DP` : Debug + Profiling + Stack check `DT` : Debug + Profiling + Stack check + Trace

Example

`libos430xl_SP.a` 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 Heap management

The heap management functions (specifically `malloc()`) included with the standard libraries that are provided by `Mitto Systems`' msp430-gcc will check for heap/stack-collisions.

To do so, the current stack pointer gets compared to the end address of the allocated chunk of RAM. When the stack pointer value is lower than that address, the library functions will call an `exit()` function and will not return to the application.

With embOS, task stacks typically are located in memory with lower addresses than the heap section, thus calling `malloc()` from a task will immediately fail that collision check. To enable the usage of `malloc()` from tasks, the respective task stacks must be placed in memory at higher addresses than the heap section.

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 `__attribute__ ((interrupt(<IRQn>)))` automatically save & restore the registers they modify and return with RETI. The interrupt vector number has to be given specified a `IRQn`.

You can also use the following macro defined in the `iomacros.h` file:

```
#define __interrupt_vec(vec)__attribute__((interrupt(vec)))
```

Example

Simple interrupt routine:

```
void __interrupt_vec(TIMER0_A0_VECTOR) OS_ISR_Tick(void) {
  IntCnt++;
}
```

Interrupt routine calling embOS functions

```
void __interrupt_vec(TIMER0_A0_VECTOR) OS_ISR_Tick(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 uses the current stack. For more information, please refer to *Interrupt stack* on page 22.

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