embOS

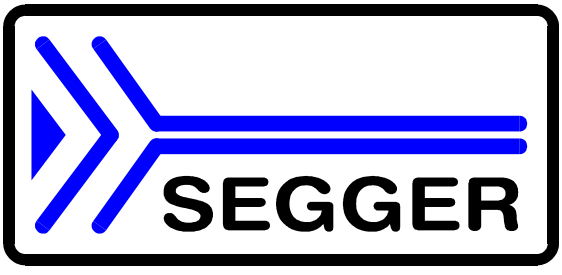
Real Time Operating System

CPU & Compiler specifics for

Texas Instruments MSP430 CPUs

and Rowley compiler for MSP430

Document Rev. 1



A product of Segger Microcontroller Systeme GmbH

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for MSP430 Real Time Operating System for the Texas Instruments MSP430 series of microcontroller using Rowley compiler for MSP430 and Rowley's Cross Studio 1.2.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** MSP430 for Rowley compiler. Before actually using ***embOS***, you should read or at least glance through this manual in order to become familiar with the software.
Chapter 2 gives you a step-by-step introduction, how to install and use ***embOS*** using Rowley CrossStudio. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use Rowley's CrossStudio debugger, because it is the easiest way to learn how to use ***embOS*** in your application.
Most of the other chapters in this document are intended to provide you with detailed information about functionality and fine-tuning of ***embOS*** for MSP430 CPUs using Rowley compiler.

2. Using *embOS* with Rowley's CrossStudio

2.1. Installation

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

In order to install it, proceed as follows:

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

Assuming that you are using Rowley's CrossStudio to develop your application, no further installation steps are required. You will find a prepared sample workspace including one start project, which you should use and modify to write your application. So follow the instructions of the next chapter 'First steps'.

You should do this even if you do not intend to use Rowley's CrossStudio for your application development in order to become familiar with ***embOS.***

If for some reason you will not work with Rowley's CrossStudio, you should:
Copy either all or only the library-file that you need to your work-directory. Also copy the hardware initialization file RTOSInitxxx.c and the ***embOS*** header file RTOS.h. 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 Rowley's CrossStudio, it may be used without the IDE using batch files or a make utility without any problem.

2.2. First steps

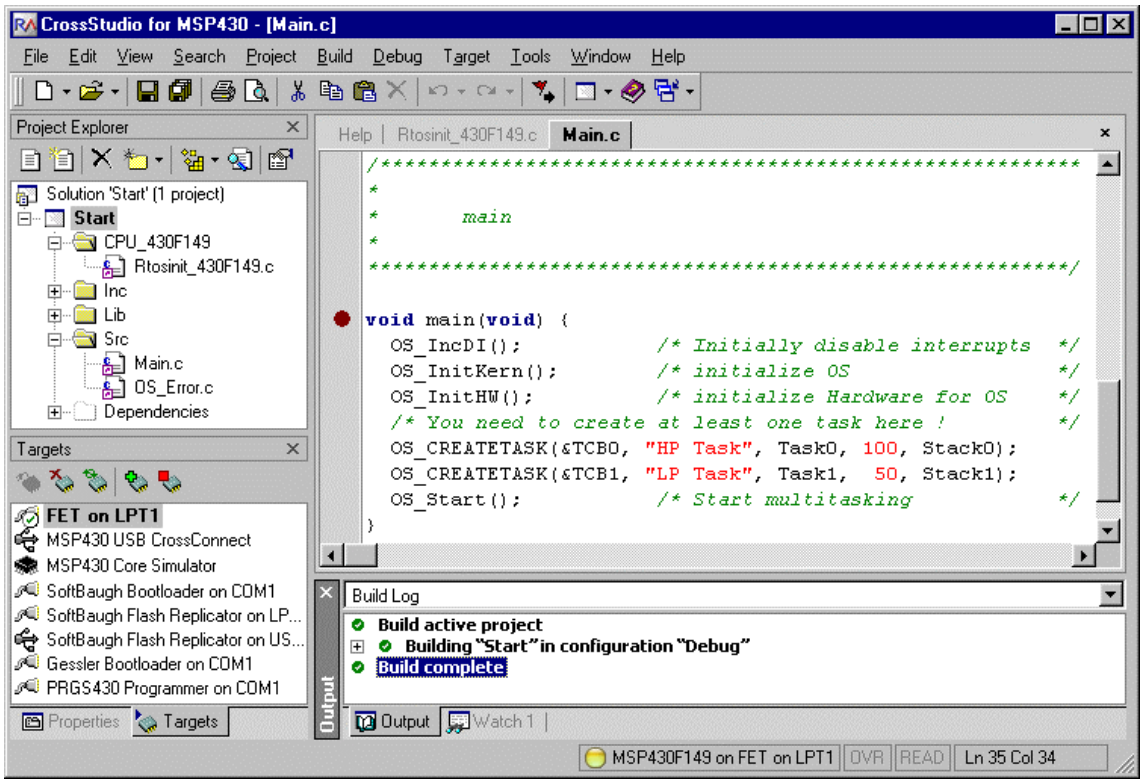
After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample workspace and start project for MSP430 CPUs and it is a good idea to use this as a starting point of all your applications.

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’ from your ***embOS*** distribution into your work directory.
- Clear the read only attribute of all files in the new ‘Start’-folder in your working directory.
- Open the folder ‘Start’.
- Open the sample workspace ‘Start.hzp.’. (e.g. by double clicking it)
- Build the start project

Further examples in this manual show the configuration for the built-in Debugger using FET Target for debugging in CrossStudio. Other targets are supported by selecting another “Target” and is similar and looks the same.

After building the start project your screen should look like follows:



For latest information you should open the ReadMe.txt which is part of your ***embOS*** distribution

2.3. The sample application Main.c

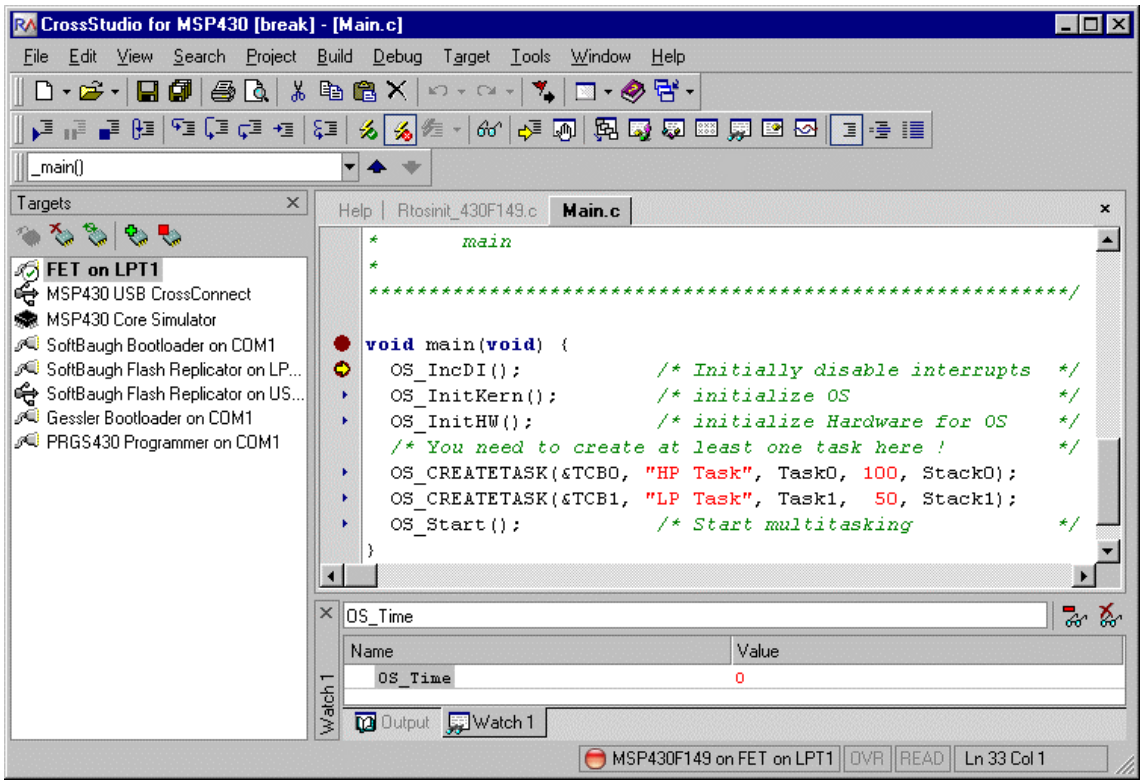
The following is a printout of the sample application main.c. It is a good starting-point for your application.
What happens is easy to see:
After initialization of ***embOS;*** two tasks are created and started
The two tasks are activated and execute until they run into the delay, then suspend for the 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 Task0(void) {
  while (1) {
    OS_Delay (10);
  }
}

void Task1(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", Task0, 100, Stack0);
  OS_CREATETASK(&TCB1, "LP Task", Task1,  50, Stack1);
  OS_Start();           /* Start multitasking            */
}
```

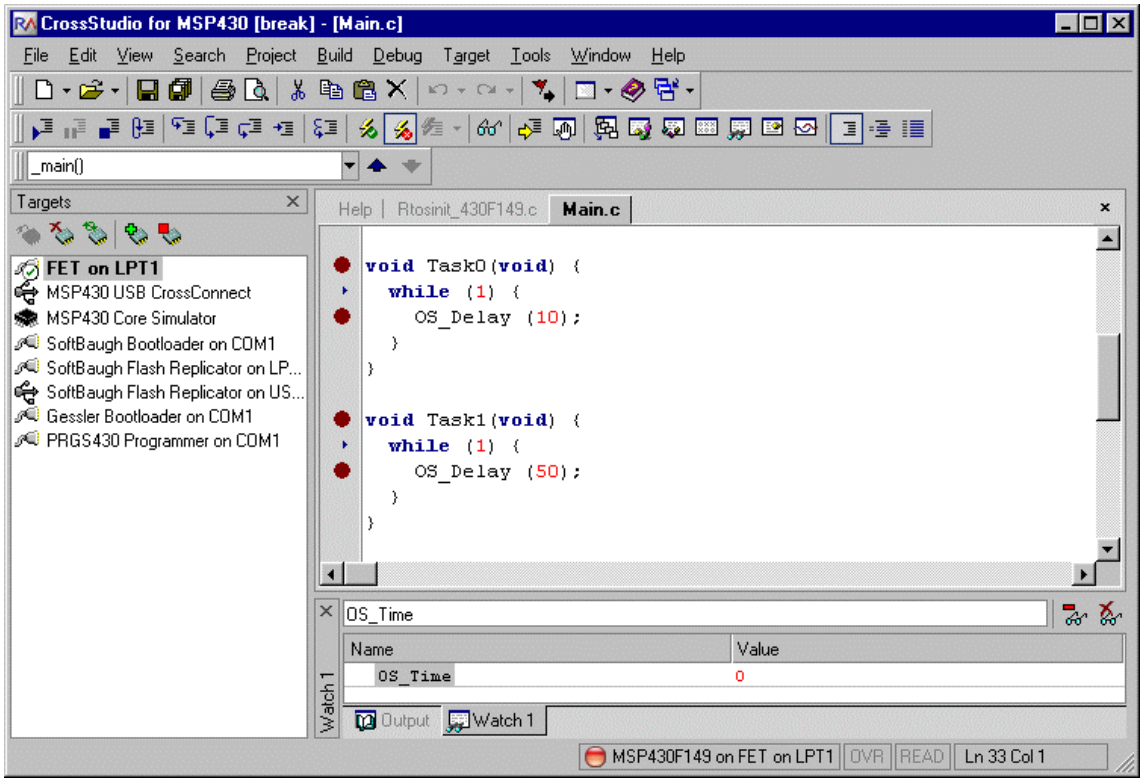
2.4. Stepping through the sample application Main.c using Debugger

When starting the debugger, you will usually see the main function (very similar to the screenshot below). If you may look at the startup code, you have to set a breakpoint at main, which is set by default. Now you can step through the program.
`OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the ***embOS*** library; you can therefore only step into it in disassembly mode. It initializes the relevant OS-Variables. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.

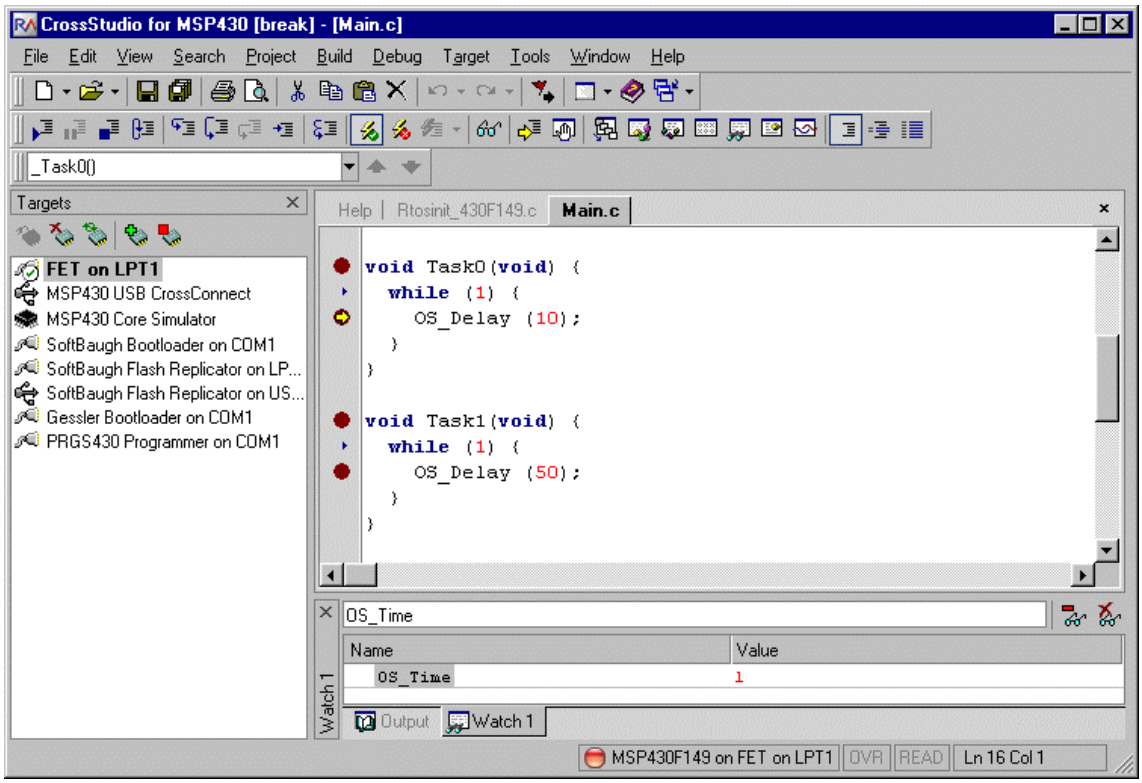
`OS_InitHW()` is part of RTOSInit.c and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.
`OS_COM_Init()` in `OS_InitHW()` is optional. It is required if embOSView shall be used. In this case it should initialize the UART used for communication.
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.



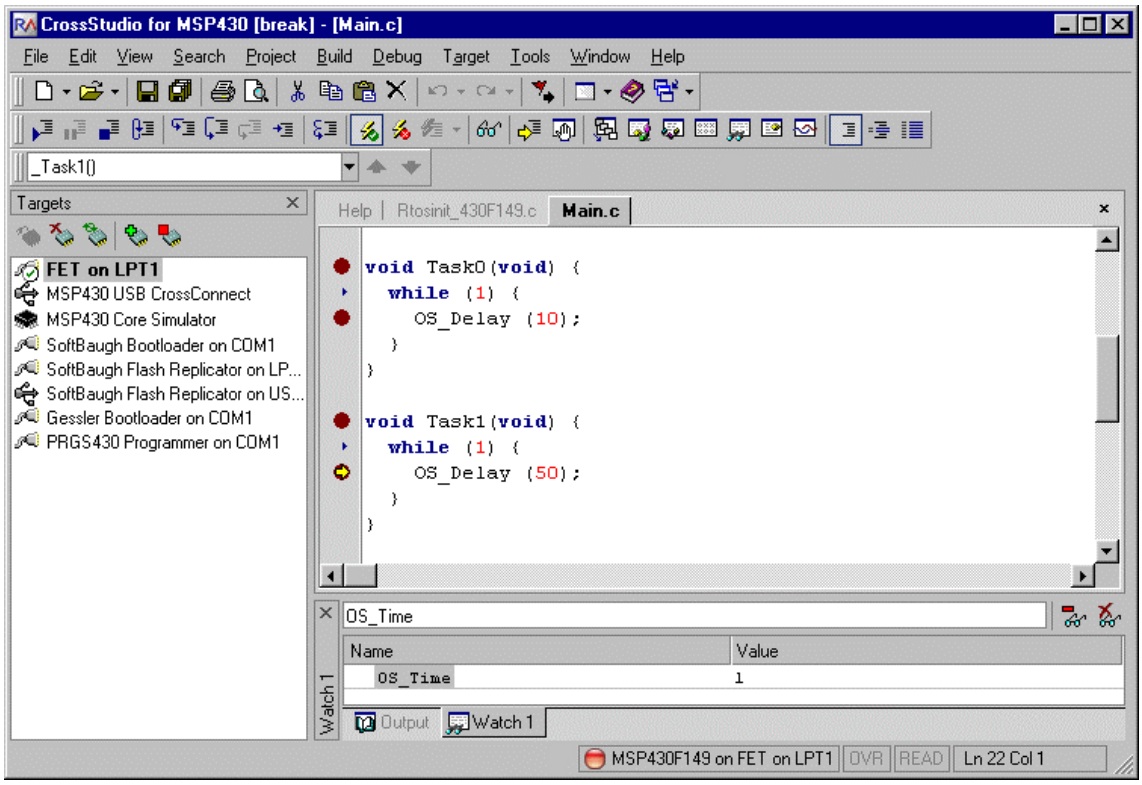
Before you step into `OS_Start()`, you should set breakpoints in the two tasks:



When you step over `OS_Start()`, the next line executed is already in the highest priority task created. (you may also step into `OS_Start()`, then stepping through the task switching process in disassembly mode). In our small start program, `Task0()` is the highest priority task and is therefore active.

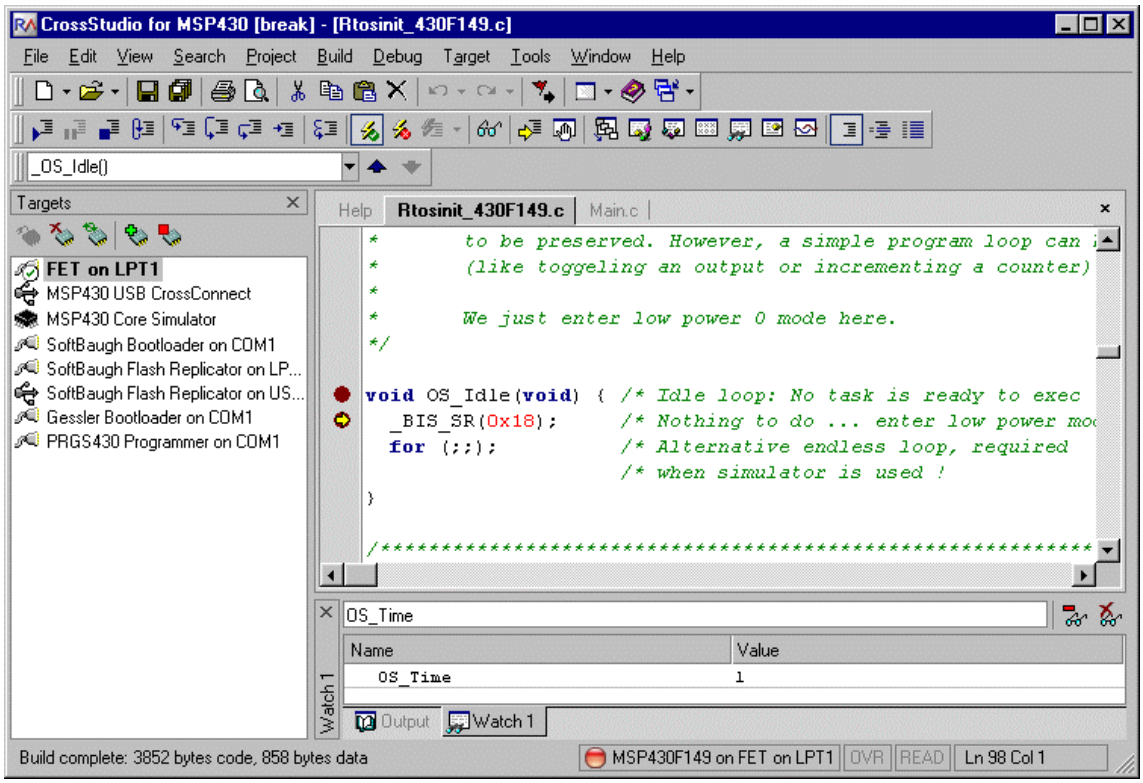


If you continue stepping, you will arrive in the task with the lower priority:

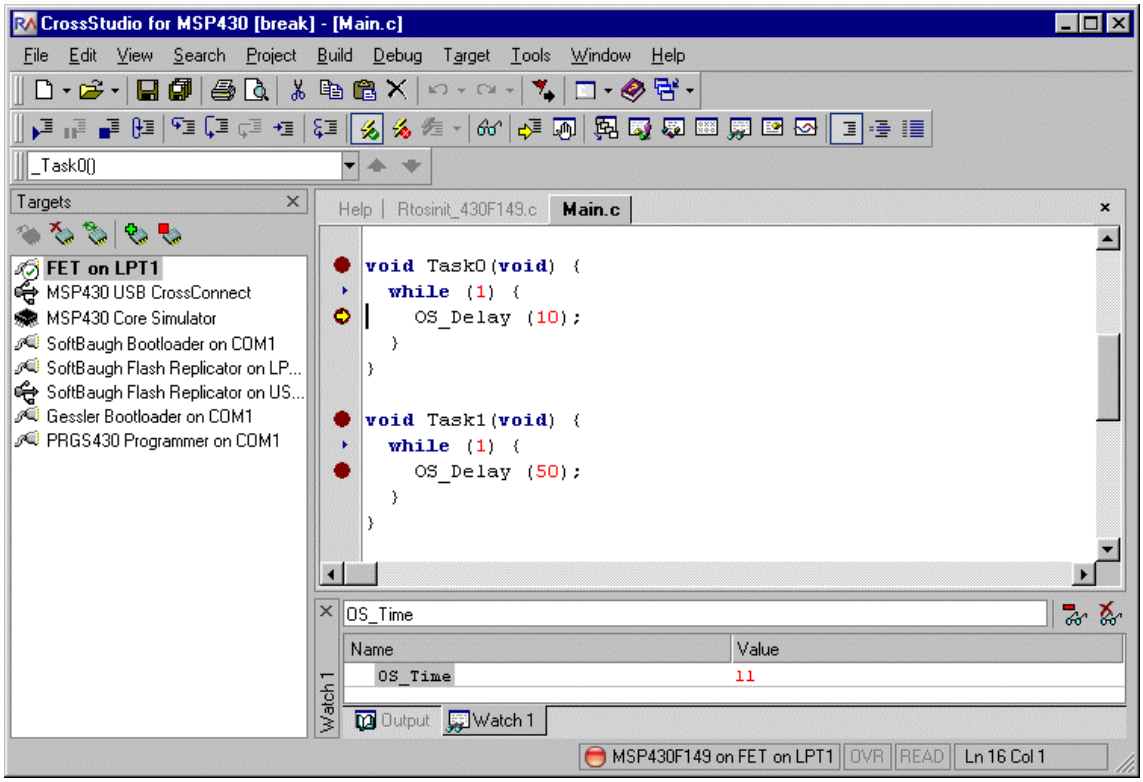


Continuing to step through the program, there is no other task ready for execution. ***embOS*** will suspend Task1 and switch to the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

`OS_Idle()` is found in RTOSInit.c:



If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay. Coming from `OS_Idle()`, you should execute the 'Go' command to arrive at the highest priority task after its delay is expired. This can be seen at the system variable OS_Time:



3. Build your own application

To build your own application, you should start with the sample start project. This has the advantage, that all necessary files are included and all settings for the project are already done.

3.1. Required files for an *embOS* application

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

- **RTOS.h** from sub folder Inc\
  This header file declares all ***embOS*** API functions and data types and has to be included in any source file using embOS functions.
- **RTOSInit_*.c** from CPU specific subfolder CPU_*\.
  It contains hardware dependent initialization code for ***embOS*** timer and optional UART for embOSView.
- One ***embOS*** **library** from the Lib\ subfolder
- **OS_Error.c** from subfolder Src\
  The error handler is used if any library other than Release build library is used in your project.

When you decide to write your own startup code, please 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 call of `OS_InitKern()` and `OS_InitHW()` prior any other ***embOS*** functions except `OS_IncDI()` are called.

3.2. Select a start project

embOS comes with one start project which includes different configurations for different output formats or debug tools. The start project was built and tested with MSP430F149 CPU. For various CPU variants there may be modifications required in RTOSInit.c.

3.3. Add your own code

For your own code, you may add a new group to the project.
You should then modify or replace the main.c source file in the subfolder src\.

3.4. Change library mode

For your application you may wish to choose an other library. For debugging and program development you should use an ***embOS*** -debug library. For your final application you may wish to use an ***embOS*** -release library.
Therefore you have to select or replace the ***embOS*** library in your project or target:

- in the Lib group, exclude all libraries from build, except the one which should be used for your application.

Finally check project options about library mode setting according library mode used. Refer to chapter 4 about the library naming conventions to select the correct library and library mode specific define.

4. Project and compiler specifics

4.1. Available libraries

The ***embOS*** library files are located in the subfolder 'Lib' of the start project folder.
To use ***embOS***, one library has to be included to your project. The files to use depend on additional error check possibilities wished to be used.

The naming convention for library files is as follows:

rtos<LIBRARYTYPE>.hza

<LIBRARYTYPE> specifies the type of ***embOS*** -library:

- **R** stands for Release build library.
- **S** stands for Stack check library, which performs stack checks during runtime.
- **SP** stands for Stack check and Profiling library, which performs stack checking and additional runtime (Profiling) calculations
- **D** stands for Debug library which performs error checking during runtime.
- **DP** stands for Debug and Profiling library which performs error checking and additional Profiling during runtime.
- **DT** stands for Debug and Trace library which performs error checking and additional Trace functionality during runtime.

All libraries were built with standard settings also used in our start project.

Example:

rtosSP.hza is the ***embOS*** library with **S**tack check and **P**rofiling functionality.
It is located in the Start\lib\ subdirectory.

For MSP430, the following libraries are available:

Library type	Library	#define
Release	rtosR.hza	OS_LIBMODE_R
Stack-check	rtosS.hza	OS_LIBMODE_S
Stack-check + Profiling	rtosSP.hza	OS_LIBMODE_SP
Debug	rtosD.hza	OS_LIBMODE_D
Debug + Profiling	rtosDP.hza	OS_LIBMODE_DP
Debug + Profiling + Trace	rtosDT.hza	OS_LIBMODE_DT

Ensure that the define, according to the library type used, is set as compiler option in your project.
Please check "Project | Properties | Preprocessor | Preprocessor Definitions".

5. Stacks

5.1. Task stack for MSP430

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location.
The stack-size required is the sum of the stack-size of all routines plus basic stack size.
The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by ***embOS*** -routines.
For the MSP430, the minimum stack size is about 30 bytes.
As MSP430 does not support its own interrupt stack, please note, that interrupts also run on task stacks. We recommend at least a minimum task stack size of 70 bytes. You may use embOSView to analyze the total amount of task stack used in your application.

5.2. System stack for MSP430

The system stack size required by ***embOS*** is about 30 bytes (60 bytes in. profiling builds) However, since the system stack is also used by the application before the start of multitasking (`main()`, before the call to `OS_Start()`), and because software-timers and ***embOS*** internal scheduling functions also use the system-stack, the actual stack requirements depend on the application.
The system stack ends at the end of the RAM. This way, all available RAM can be used as system stack.
For builds with stack checking, the amount of memory filled with the control byte is set in RTOSInit_*.c, (`OS_GetSysStackSize()`).

5.3. Interrupt stack for MSP430

Unfortunately MSP430 CPUs do not support a separate interrupt stack pointer. Interrupts use the stack of the running application. Therefore interrupts occupy additional stack space on every task and on the system stack. Current version of ***embOS*** does not support a separate interrupt stack.

5.4. Stack specifics of the MSP430 family

MSP430 family of microcontroller can address up to 64KB of memory. Because the stack-pointer can address the entire memory area, stacks can be located anywhere in RAM.

6. MSP430 clock specifics

MSP 430 CPUs offer various options for CPU and peripheral clock generation. You may have to modify the ***embOS*** timer initialization function `OS_InitHW()` in RtosInit_*.c to fit your application.
The sample `OS_InitHW()` routine in RTOSInit_430F149.c uses internal RC oscillator and DCO as CPU clock and selects this clock as source for ***embOS*** timer.

6.1. *embOS* timer clock source

embOS timer may be driven from different clock sources. Per default, MCLK is used as clock source for timer. `OS_IntHW()` initializes RC oscillator and DCO.

Using RC oscillator:

Usage of RC oscillator and DCO has the advantage that no additional hardware is required. The disadvantage is, that exact frequency is nor guaranteed neither precise.
The frequency of MCLK has to be examined and `OS_FSYS` has to be adjusted to fit your application.

Using external crystal:

Usage of an external Main clock generator crystal has the advantage, that frequency is stable and precise. OS_InitHW() has to be modified to initialize crystal oscillator. Check OS_FSYS and set a value that corresponds to your crystal frequency.

6.2. Clock for UART

When using embOSView to analyze target system, a USART has to be initialized for serial communication. Per default, communication to embOSView is enabled and uses USART0.
Different clock sources for baudrate generation may be used. Per default, `OS_COM_Init()` initializes auxiliary clock oscillator as clock source for USART. This ensures accurate baudrate up to 9600 baud.

Using ACLK for UART:

ACLK as source for UART has the advantage, that baudrate is accurate up to 9600 BAUD.

Using MCLK for UART:

MCLK may be used as clock source for baudrate generator. Higher baudrates are possible. When RC oscillator is used for MCLK generation, baudrate may be instable. When crystal is used for MCLK generation, baudrate is accurate and stable. To use MCLK as source for baudrate generator, define `OS_BAUDSRC_MCLK=1` as project option. Baudrate generator settings will then be calculated derived from `OS_BAUDRATE` and `OS_FSYS`.

7. Interrupts

7.1. What happens when an interrupt occurs?

- The CPU-core 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
- ISR : user-defined functionality
- ISR : restore registers
- ISR: Execute RETI command, restoring PC, Flags and continue interrupted program
- For details, please refer to Texas Instruments users manual.

7.2. 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 inside the brackets.
For a detailed description on how to define an interrupt routine in "C", refer to the Rowley's C-Compiler reference guide.

Example

"Simple" interrupt-routine

```
void IntHandlerTimer(void) __interrupt[12] {
  IntCnt++;
}
```

Interrupt-routine calling ***embOS*** functions

```
void IntHandlerTimer(void) __interrupt[12] {
  OS_EnterInterrupt();  /* Inform embOS that interrupt function is running */
  IntCnt++;
  OS_PutMailCond(&MB_Data, &IntCnt);
  OS_LeaveInterrupt();
}
```

`OS_EnterInterrupt()` has to be the first function called in an interrupt handler using ***embOS*** functions, when nestable interrupts are not required. `OS_LeaveInterrupt()` has to be called at the end the interrupt handler then. If interrupts should be nested, use `OS_EnterNestableInterrupt()` / `OS_LeaveNestableInterrupt()` instead.

7.3. Interrupt-stack

Since MSP430 CPUs 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.
Current version of ***embOS*** for MSP430 does not support extra interrupt stack-switching in an interrupt routine.
The routines `OS_EnterIntStack()` and `OS_LeaveIntStack()` are supplied for source compatibility to other processors only and have no functionality.

8. Low-Power Modes

Usage of Low-Power modes is one possibility to save power consumption during idle times. If required, you may modify the `OS_Idle()` routine, which is part of the hardware dependent module RtosInit_*.c to enter a Low-Power mode.
Please do not enter a Low-Power mode which stops the ***embOS*** timer, as this would stop time scheduled task activations.
If ***embOS*** timer is driven from main clock, LPM0 or LPM1 may be selected during `OS_Idle()`. When ACLK is used for ***embOS*** timer, LPM2 or LPM3 may also be used.

9. Technical data

9.1. 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 values in the table are for the far memory model and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1208	28
Event-management	< 200	---
Mailbox management	< 550	---
Single-byte mailbox management	< 300	---
Resource-semaphore management	< 250	---
Timer-management	< 250	---
Add. Task	---	18
Add. Semaphore	---	4
Add. Mailbox	---	12
Add. Timer	---	10
Power-management	---	---

10. Files shipped with *embOS* MSP430 Rowley

Directory	File	Explanation
root	`*.pdf`	Generic API and target specific documentation
root	`Release.html`	Release notes of ***embOS*** MSP430
root	`embOSView.exe`	Utility for runtime analysis, described in generic documentation
Start\	`Start.hzp`	Sample Project for Rowley CrossStudio
Start\	`Start.hzs`	Sample Solution (workspace) for MSP430 CPUs
Start\Inc\	`RTOS.h`	To be included in any file using ***embOS*** functions
Start\lib\	`rtos*.hza`	***embOS*** libraries
Start\Src\	`main.c`	Frame program to serve as a start
Start\Src\	`OS_Error.c`	embOS error handler used in stack check and debug builds
Start\CPU_*\	`RtosInit_*.c`	Target CPU specific hardware initialization; can be modified

11. Index