embOS

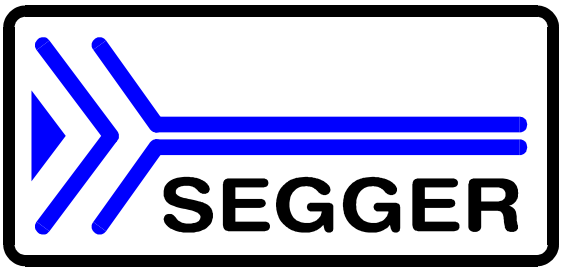
Real Time Operating System

CPU & Compiler specifics for

CR16C

using IAR Compiler and winIDEA

Document Rev. 2



A product of Segger Microcontroller Systeme GmbH

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** Real Time Operating System for the National Semiconductor CR16C micro controllers using *IAR*.

1.1. How to use this manual

This manual describes all CPU and compiler specifics of ***embOS*** using National Semiconductor CR16C micro controllers with *IAR*. Before actually using ***embOS***, you should read or at least glance through this manual in order to become familiar with the software.
Chapter 2 gives you a step-by-step introduction, how to install and use ***embOS*** for the National Semiconductor CR16C micro controllers using *IAR*. If you have no experience using ***embOS***, you should follow this introduction, because it is the easiest way to learn how to use ***embOS*** in your application.
Most of the other chapters in this document are intended to provide you with important detailed information about functionality and fine-tuning of ***embOS*** for the National Semiconductor CR16C micro controllers using *IAR*.

2. Using ***embOS*** with IAR

The following chapter describes how to start with and use ***embOS*** for National Semiconductor CR16C and IAR compiler. You should follow these steps to become familiar with ***embOS*** for National Semiconductor CR16C micro controllers using *IAR*.

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

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

If for some reason you will not work with the project manager, 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 *IAR*, it may be used without the *IAR* using batch files or a make utility without any problem.

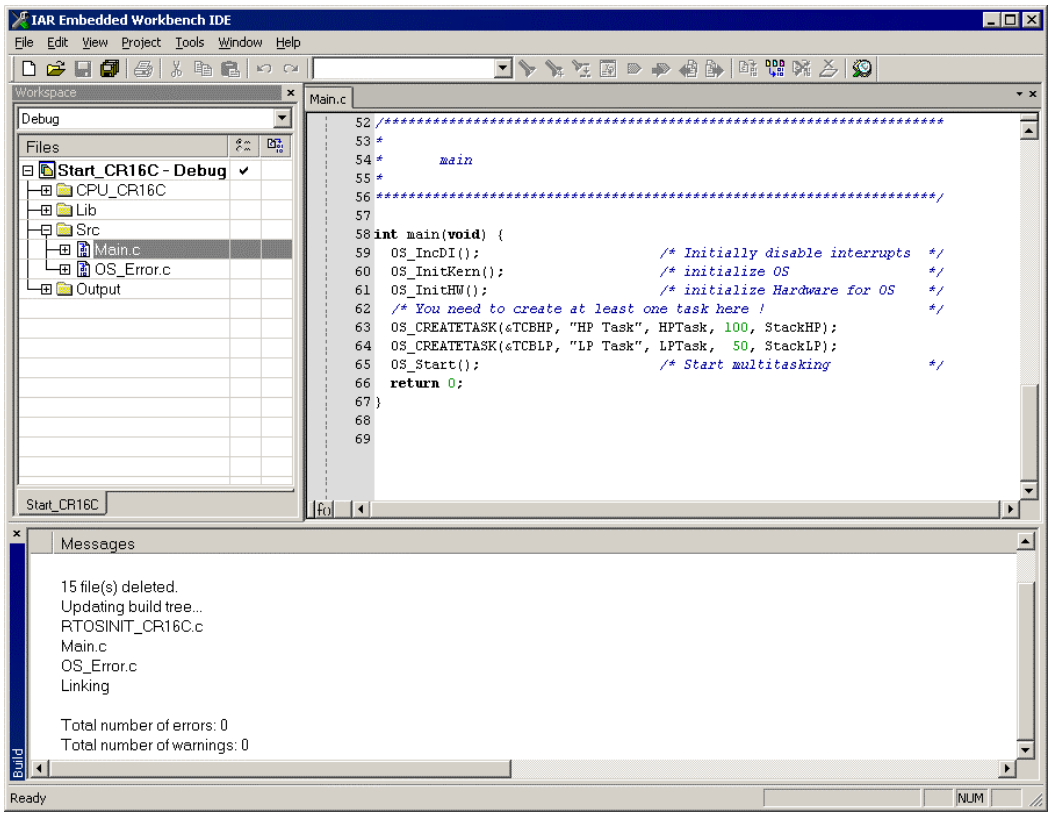
2.2. First steps

After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start workspace and a project and it is a good idea to use this as a starting point of all your applications.
Your ***embOS*** distribution contains one folder “Start“ which contains the sample start workspace and project and every additional files used to build your application.

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

- Create a work directory for your application, for example c:\work
- Copy the whole folder ‘Start’ which is part of your ***embOS*** distribution into your work directory
- Clear the read only attribute of all files in the new ‘start’ folder.
- Open the sample workspace start\Start_CR16C.eww with *IAR* Embedded Workbench (e.g. by double clicking it)
- Build the start project

Your screen should look like follows:



For latest information you should open the file start\ReadMe.txt.

2.3. The sample application Main.c

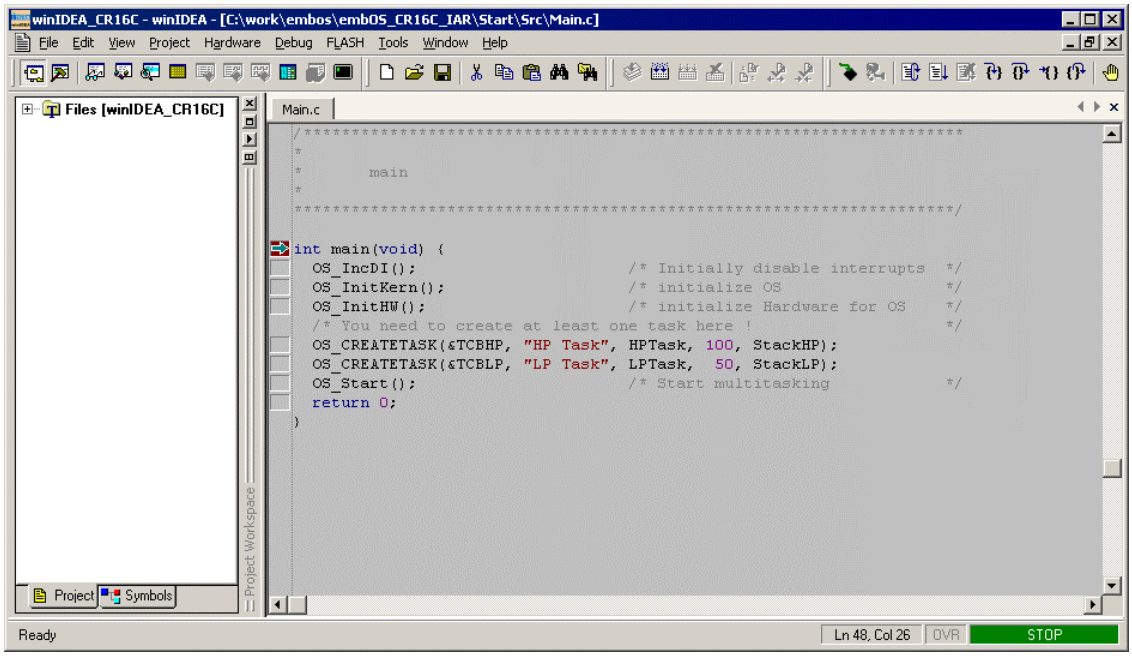
The following is a printout of the sample application main.c. It is a good starting-point for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS;*** two tasks are created and started
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
/*********************************************************
*          SEGGER MICROCONTROLLER SYSTEME GmbH
*   Solutions for real time microcontroller applications
**********************************************************
File    : 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            */
}
```

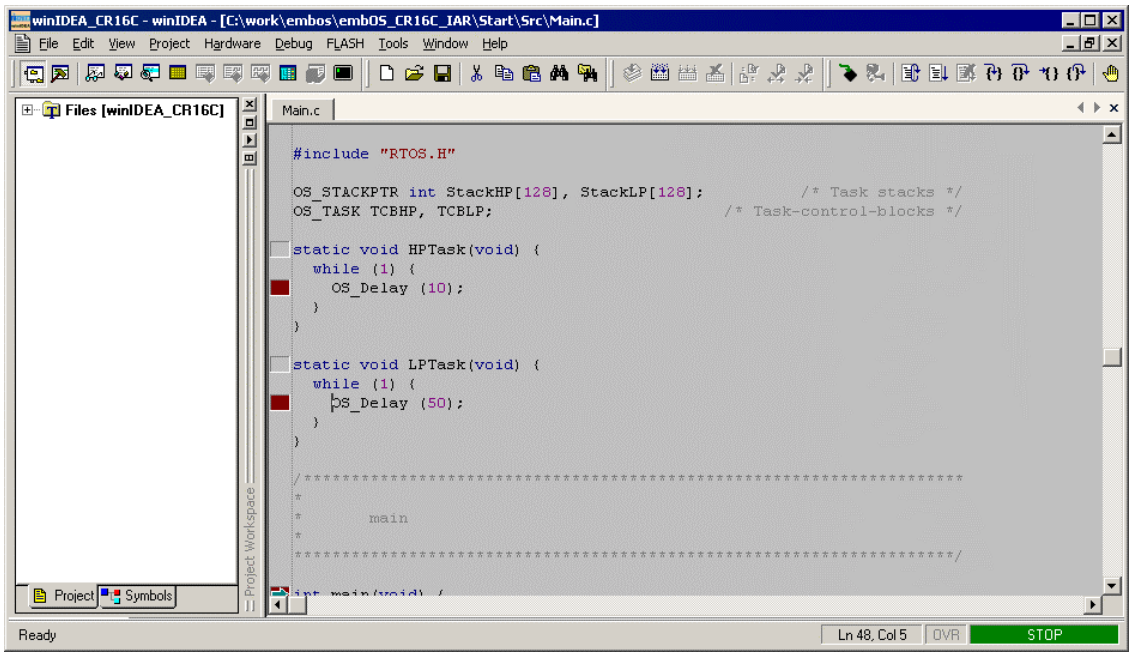
2.4. Stepping through the sample application using winIDEA

When starting the winIDEA debugger, you will usually see the main function (very similar to the screenshot below). In some debuggers, you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.
`OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the ***embOS*** library; you can therefore only step into it in disassembly mode. It initializes the relevant OS-Variables. Because of the previous call of `OS_IncDI()`, interrupts are not enabled during execution of `OS_InitKern()`.
`OS_InitHW()` is part of RTOSInit_CR16C.c and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the timer-tick-interrupt for ***embOS.*** Step through it to see what is done.

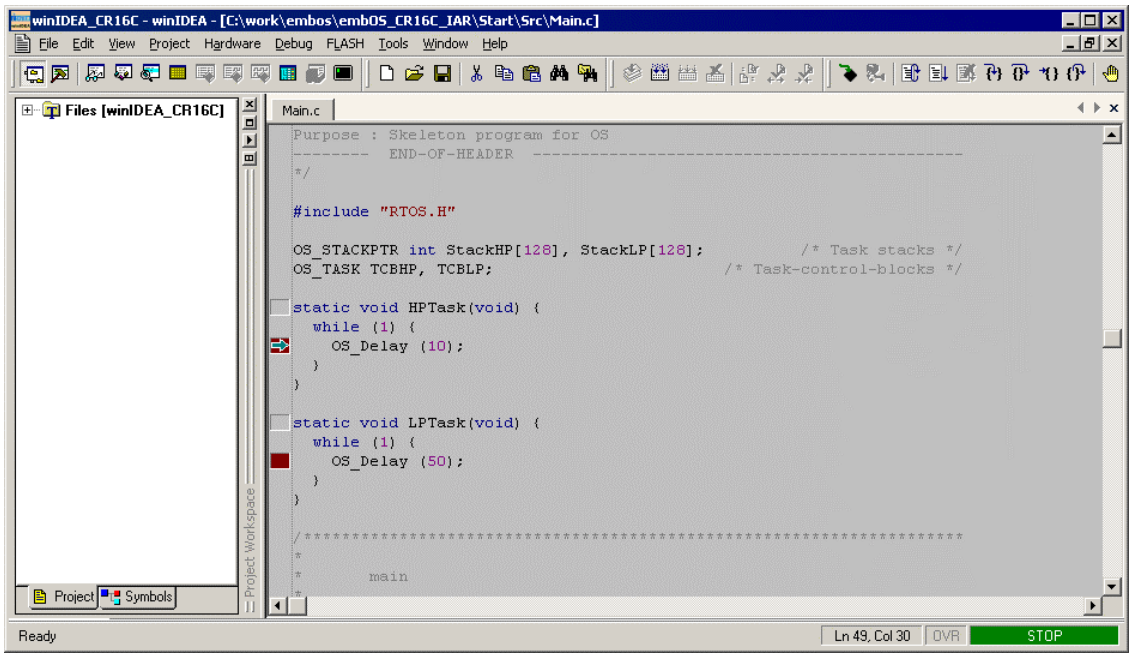
OS_Start() should be the last line in main, since it starts multitasking and does not return.



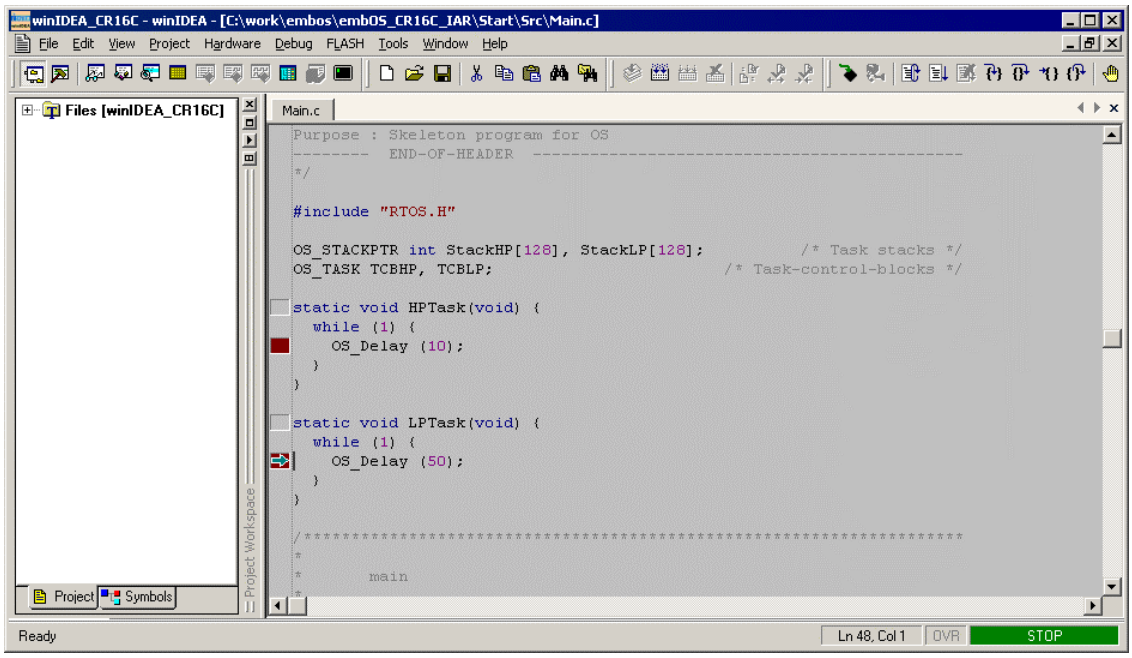
Before you step into OS_Start(), you should set two break points in the two tasks as shown below.



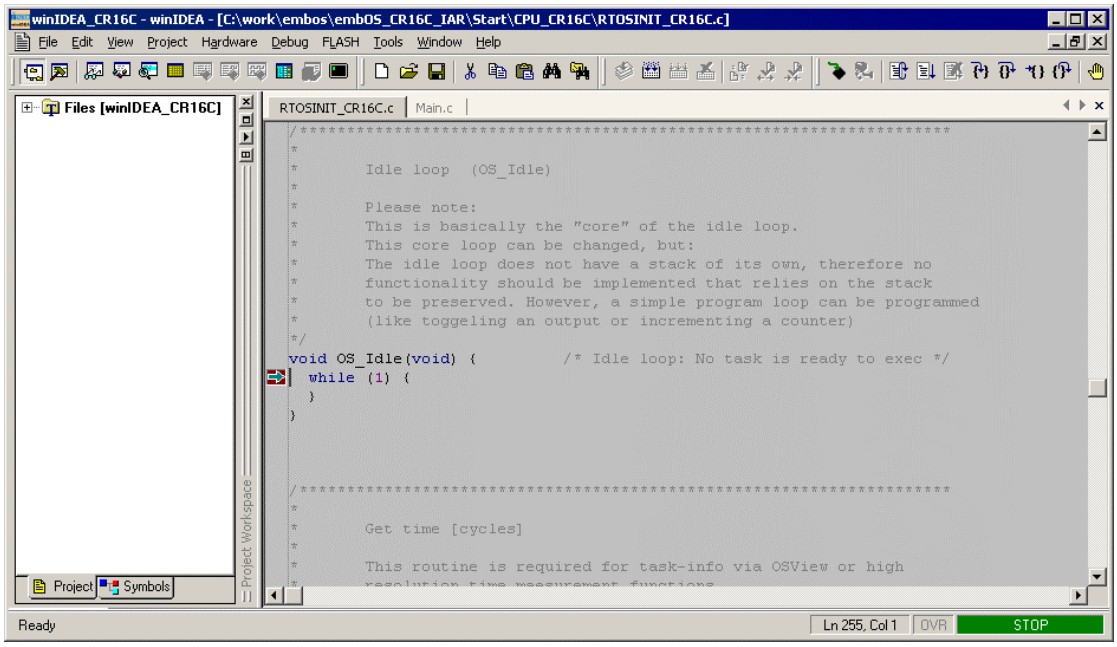
As OS_Start() is part of the *embOS* library, you can step through it in disassembly mode only. You may press GO, step over OS_Start(), or step into OS_Start() in disassembly mode until you reach the highest priority task.



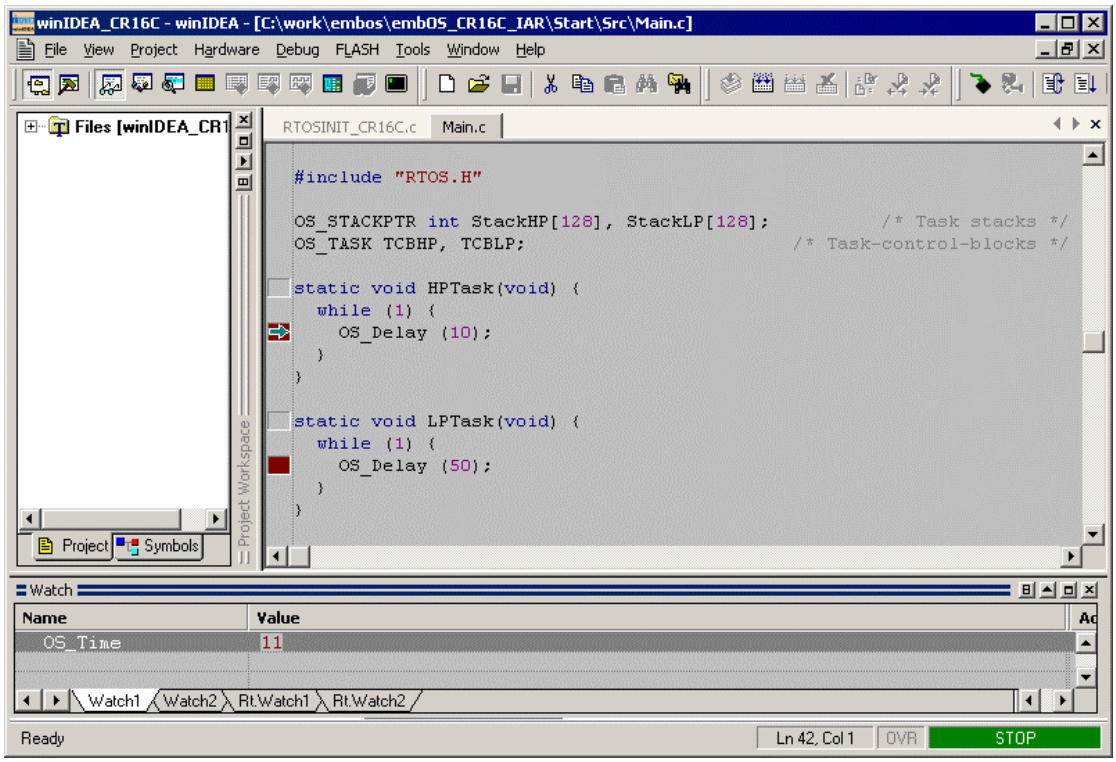
If you continue stepping, you will arrive in the task with lower priority:



Continuing to step through the program, there is no other task ready for execution. ***embOS*** will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).
You will arrive there when you step into the `OS_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit_CR16C.c`. You may also set a breakpoint there before you step over the delay in Task1.



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, Task0 continues operation after expiration of the 10 ms delay.



3. Build your own application

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

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

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

- **RTOS.h** from sub folder Inc\
  This header file declares all ***embOS*** API functions and data types and has to be included in any source file using ***embOS*** functions.
- **RTOSInit_CR16C.c** from the CPU subfolder.
  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.
- Additional low level init code may be required according to CPU.

When you decide to write your own startup code or use a __low_level_init function, please ensure that non initialized variables are initialized with zero, according to “C” standard. This is required for some ***embOS*** internal variables.
Also ensure, that main is called with CPU running in supervisor or system mode.
Your main() function has to initialize ***embOS*** by call of `OS_InitKern()` and `OS_InitHW()` prior any other ***embOS*** functions are called.
You should then modify or replace the main.c source file in the subfolder src\.

3.2. Change library mode

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

- To replace a library, first delete the actual library from the library files in the project explorer. After that you add the new library to the library files.
- Check and set the appropriate OS_LIBMODE_* define as preprocessor option in the compiler options.

4. IAR Compiler specifics

4.1. CPU modes

embOS supports nearly all memory model combinations that National Semiconductor CR16C supports.

4.2. Available libraries

embOS for National Semiconductor CR16C for IAR compiler is shipped with 24 different libraries, one for each code memory model / data memory model combination.
The libraries are named as follows:

oscr16c<c><d><LibMode>.a

Parameter	Meaning	Values
C	Specifies the data memory model	m: medium
		l: large
D	Specifies if indexed addressing mode is used	n: no indexed mode
		i: indexed addressing mode
LibMode	Library mode	R: Release
		S: Stack check
		D: Debug
		SP: Stack check + profiling
		DP: Debug + profiling
		DT: Debug + trace

Example:
oscr16cmir.r45 is the library for a project using the medium data memory model, indexed addressing mode and release build library type.

oscr16clnsp.r45 is the library for a project using the large data memory model, no indexed addressing mode and stack check plus profiling build library type.

5. Stacks

5.1. Task stack for CR16C CPUs

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location that can be used as stack by the CR16C CPU.
As CR16C CPUs have a 16 bit stack pointer, the whole internal memory area can be used as task stack.
Please note, that the task stacks have to be aligned at EVEN addresses. To ensure proper alignment, implement task stack as array of int.
The stack-size required for tasks 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 CR16C, this minimum task stack size is about 28 bytes. We recommend at least a minimum of 128 bytes for task stacks.

5.2. System stack for CR16C

The minimum system stack size required by ***embOS*** is about 144 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 project settings.

5.3. Interrupt stack for CR16C

The CR16C CPUs have a separate interrupt stack pointer. The interrupt service routines use the interrupt stack only for the program status word, the return address and a copy of the task stack pointer if you use OS_EnterIntStack() and OS_LeaveIntStack(). All other data is stored on the system stack.

6. Interrupts

6.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the processor interrupt priority level is below to the interrupt priority level, the interrupt is executed
- the CPU saves several registers on the interrupt stack.
  These are:
  PC (program counter)
  PSR (program status register)
- the CPU calls the interrupt service routine, that belongs to the received interrupt.
- User defined functionality in interrupt service routine
- Execute *retx* to return from interrupt, restoring all saved registers
- For details, please refer to CR16C user manual

6.2. Defining interrupt handlers in "C"

Routines defined with the keywords `__interrupt` automatically save & restore the registers they modify and return with RETX.
For a detailed description on how to define an interrupt routine in "C", refer to the CR16C C-Compiler's user's guide.

Example

"Simple" interrupt-routine

```
#pragma vector=(UART0TX_IRQ+MASKINTS_START)
__interrupt void OS_Uart0TxHandler(void) {
  OS_EnterInterrupt();
  OS_EnterIntStack();
  if (OS_OnTx()) {      //  No more characters to send?
     U0ICTRL &= ~0x20;  //  Disable Tx Interrupt
  }
  OS_LeaveIntStack();
  OS_LeaveInterrupt();
}
```

6.3. Interrupt Vector Table

The interrupt vector table is located in any location in ROM or RAM. The value of the *INTBASE* register points to the start address of the interrupt vector table.

6.4. Interrupt Stack

The CR16C CPUs have a separate interrupt stack pointer. The interrupt service routines use the interrupt stack only for the program status word and the return address. All other data will be stored on the system stack if you use the EnterIntStack() and LeaveIntStack() macros. The EnterIntStack() macro will add two bytes on the interrupt stack. Be sure to define your system stack size big enough, so that all nested interrupt routines can run on this stack!

6.5. Interrupt priorities

The CR16C CPU has hardware fixed interrupt priorities. They cannot be changed by software.

7. STOP / WAIT Mode

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 function OS_Idle(), which you can find in ***embOS*** module RTOSINIT_CR16C.c.

8. Technical data

8.1. 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***.
In the table below, you can find minimum RAM size for ***embOS*** resources. Please note, that sizes depend on selected ***embOS*** library mode; table below is for a release build.

embOS resource	RAM [bytes]
Task control block	28
Resource semaphore	14
Counting semaphore	6
Mailbox	16
Software timer	14

9. 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	`Start.*`	Sample workspace and project files for MPLAB IDE.
START\INC	`RTOS.H`	Include file for ***embOS***, to be included in every "C"-file using ***embOS*** -functions
START\LIB	`Rtos*.a`	***embOS*** libraries
START\SRC	`main.c`	Sample frame program to serve as a start
START\SRC	`OS_Error.c`	***embOS*** runtime error handler used in stack check or debug builds
START\CPU_CR16C	`*.*`	CPU specific hardware routines.

Any additional files shipped serve as example.

10. Index