embOS

Real-Time Operating System

CPU & Compiler specifics for ARC EM4 using MetaWare

Document: UM01080
Software Version: 5.18.3.0
Revision: 0
Date: April 10, 2024

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: April 10, 2024

Software	Revision	Date	By	Description
5.18.3.0	0	240410	TS	New software version.
5.16.1.3	0	220728	TS	Chapter “Interrupts” updated.
5.16.1.2	0	220427	TS	New software version.
5.16.1.1	0	220315	TS	Chapter “Interrupts” updated.
5.16.1.0	0	220126	TS	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.

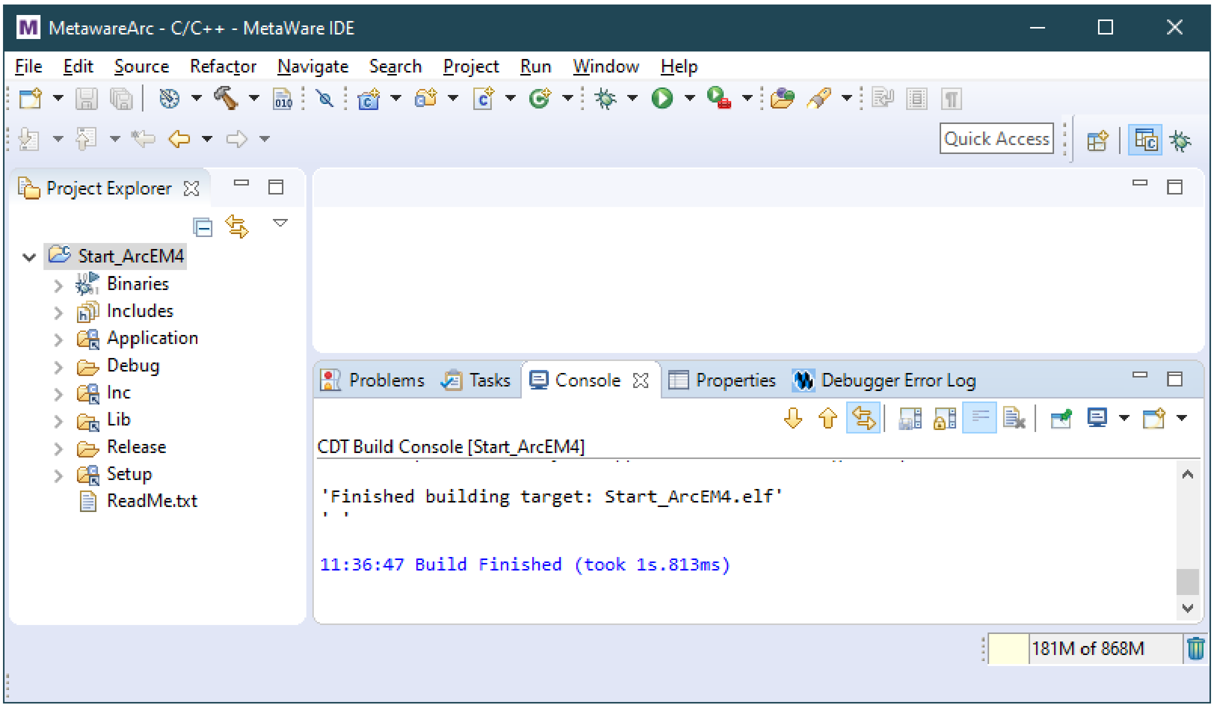
1.2 First Steps

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:



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

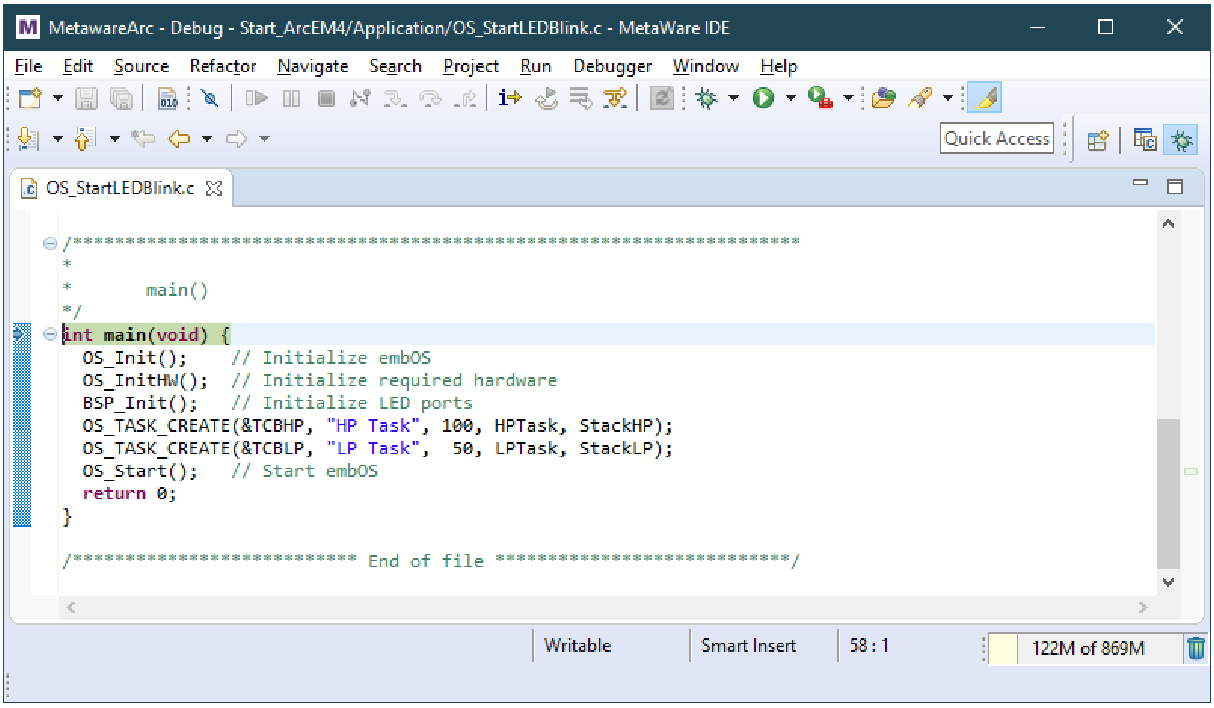
1.4 Stepping through the sample application

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.

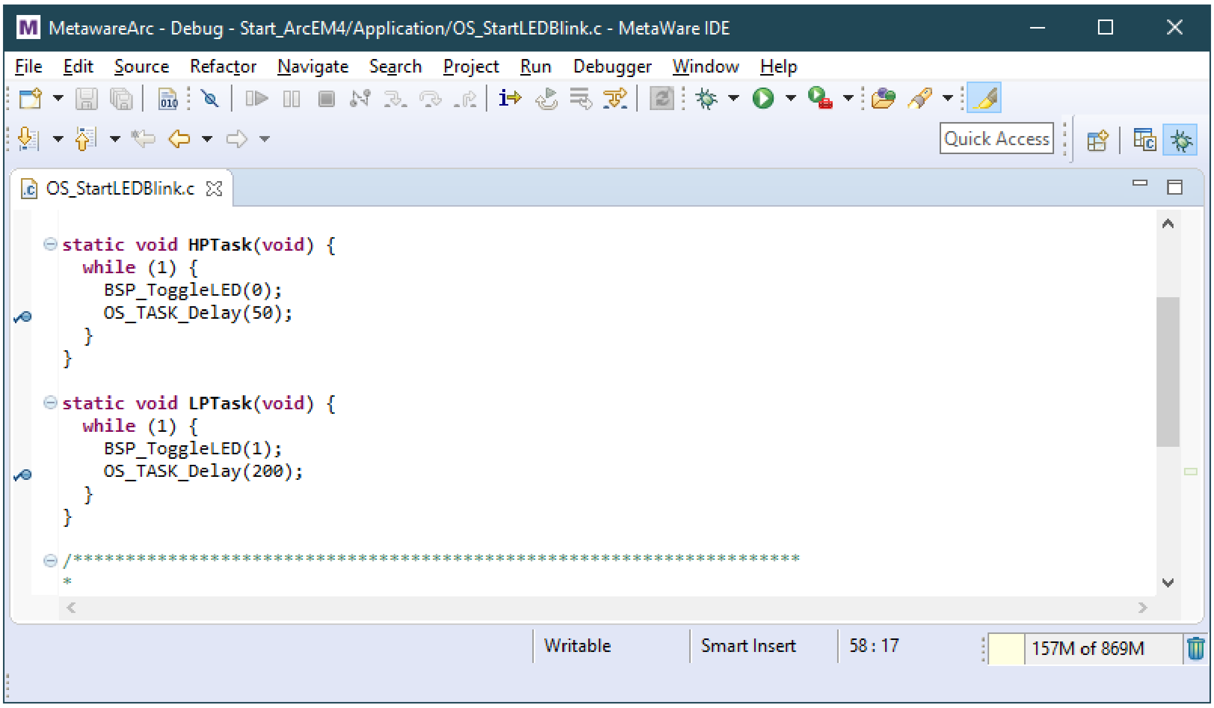
`OS_Init()` is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables.

`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.

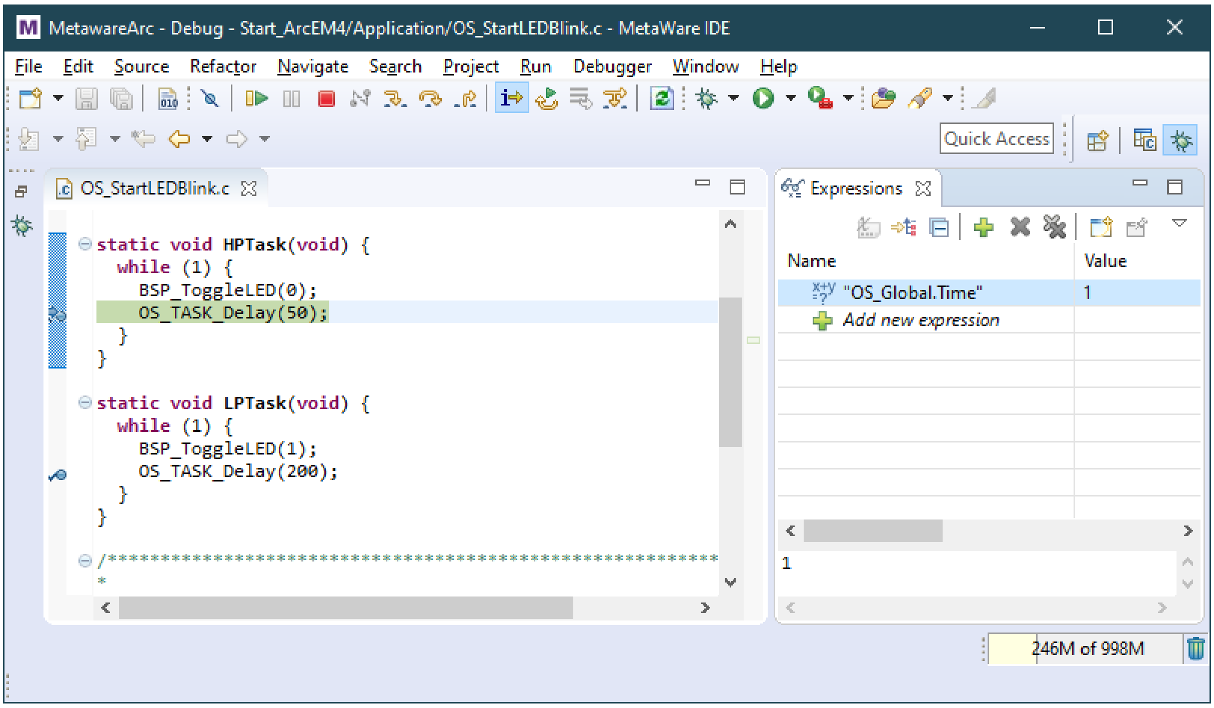


Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.

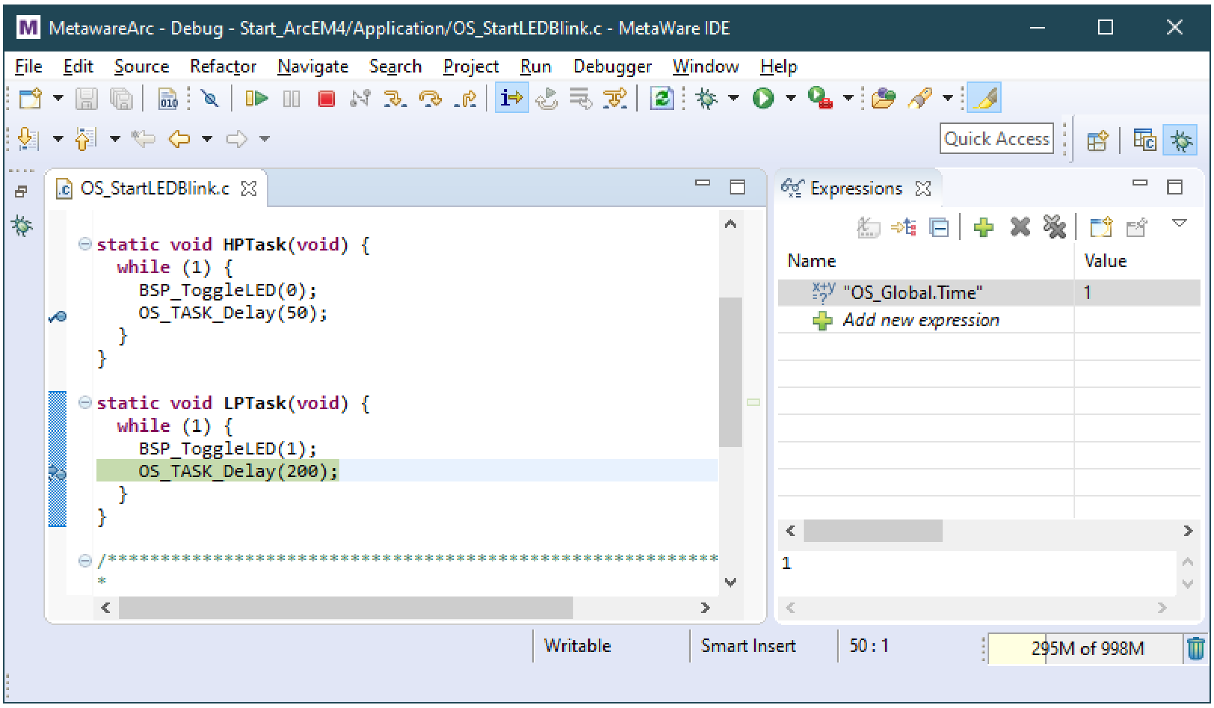


As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

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

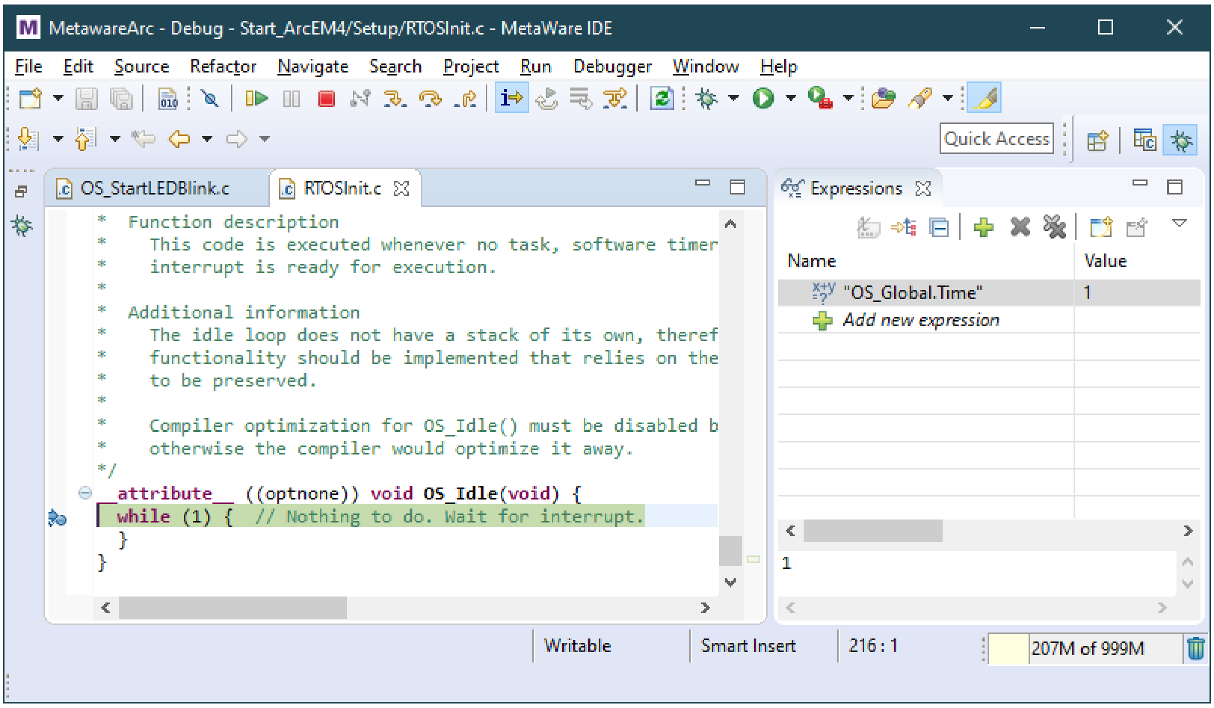


If you continue stepping, you will arrive at the task that has lower priority:



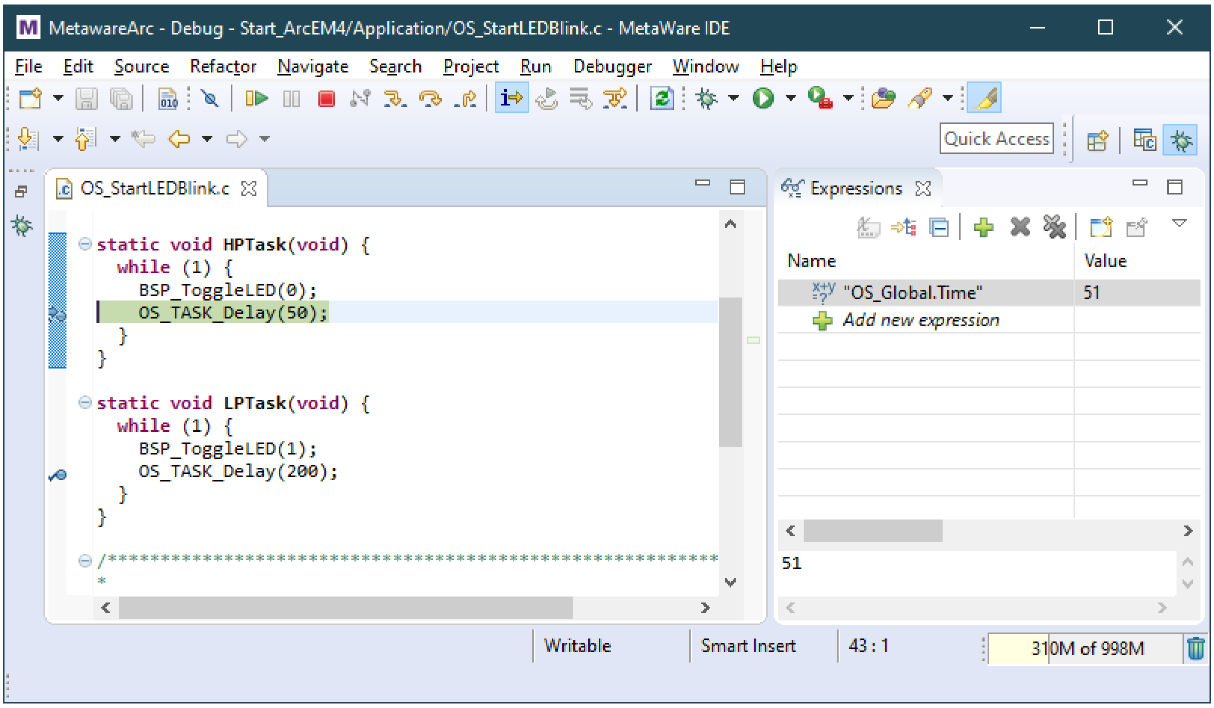
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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 built for the medium memory model.

The libraries are named as follows:

`libosem4_<LibMode>.a`

Parameter	Meaning	Values
LibMode	Specifies the 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

`libosem4_DP.a` is the library for a project using ARC EM4 core with debug and profiling support.

Chapter 4

CPU and compiler specifics

4.1 Supported cores

embOS ARC MetaWare was built for a custom specific ARC EM4 core. embOS ARC MetaWare may need to be modified to run on other ARC EM4 cores.

4.2 Thread safety

If non thread-safe functions are used from different tasks, embOS API like mutex may be used to encapsulate these functions and guarantee mutual exclusion.

The run-time libraries from DesignWare ARC MetaWare Toolset 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 ARC MetaWare Toolset is prepared to use these hook functions. Adding the file `OS_ThreadSafe.c`, which is delivered with embOS, activates the automatic thread locking functionality.

> **Note**
>
> The Metaware Compactlib run-time library (-Hcl) does not support thread-safety.

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 ARC EM4 CPUs, this minimum basic task stack size is about 128 bytes. Because any function call uses some amount of stack and every exception also pushes at least 72 bytes onto the current stack, the task stack size has to be large enough to handle one exception too. We recommend at least 512 bytes stack as a start.

5.2 System stack

The minimum system stack size required by embOS is about 256 bytes (stack check build). However, since the system stack is also used by the application before the start of multi-tasking (the call to `OS_Start()`), and because software timers and interrupt handlers also use the system stack, the actual stack requirements depend on the application. The size of the system stack can be changed in the linker file. We recommend a minimum stack size of 512 bytes.

5.3 Interrupt stack

The ARC EM4 core does not support a separate interrupt stack.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- As soon as the interrupts are enabled, the interrupt is executed.
- PC and STATUS32 register are saved on the stack.
- The interrupt service routine is started.
- Code generated by the compiler saves all scratch register.
- If the application interrupt routine caused a task switch the context switch is executed within the ISR.
- The interrupt service routine restores the scratch register and returns to the application. PC and STATUS32 are restored from the stack.

6.2 Defining interrupt handlers in C

Interrupt handlers for ARC EM4 core are written as normal C-functions which do not take parameters and do not return any value. You must use the keyword `_Interrupt` or `__attribute__((__interrupt__))` to mark the function as an interrupt handler. embOS must be informed that an interrupt routine is executed. For that purpose you must call `OS_INT_Enter()` and `OS_INT_Leave()`. Please have a look in the generic embOS manual in chapter "Interrupts" for more details.

Example

```
__attribute__((__interrupt__)) void SysTick_Handler(void) {
  _sr(TIMER_CONTROL_NH_BIT | TIMER_CONTROL_IE_BIT, TIMER0_CONTROL);
  OS_INT_Enter();
  OS_TICK_Handle();
  OS_INT_Leave();
}
```

6.3 Interrupt vector table

The file `vectors.s` includes the vector tables. Please ensure this file is included in your project and your application interrupt routines are inserted there.

6.4 Zero latency interrupts

embOS ARC MetaWare does not support zero latency interrupt.

6.5 Interrupt priorities

embOS ARC MetaWare does not support ARC EM4 interrupt priorities. It assumes there is only one interrupt priority level and interrupts are not nestable.

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	~136 bytes
Task control block	RAM	36 bytes
Software timer	RAM	20 bytes
Task event	RAM	0 bytes
Event object	RAM	16 bytes
Mutex	RAM	16 bytes
Semaphore	RAM	8 bytes
RWLocks	RAM	28 bytes
Mailbox	RAM	24 bytes
Queue	RAM	32 bytes
Watchdog	RAM	12 bytes
Fixed Block Size Memory Pool	RAM	32 bytes

