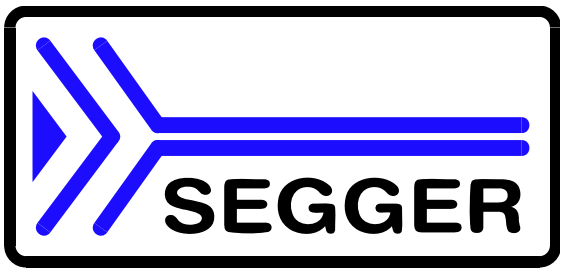
embOS

Real Time Operating System

CPU & Compiler specifics for Cortex A8 cores using CodeSourcery CodeBench IDE

Software version 3.82t
Document UM01024
Revision: 0



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER MICROCONTROLLER GmbH & Co. KG (the manufacturer) assumes no responsibility for any errors or omissions. The manufacturer makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. The manufacturer specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel.+49 2103-2878-0
Fax.+49 2103-2878-28
Email: support@segger.com
Internet: http://www.segger.com

Software and manual versions

his manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: May 31, 2011

Software	Manual	Date	By	Description
3.82t	0	110527	TS	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.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections

Table 1.1: Typographic conventions

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development-time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficent real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER developes and produces programming tools for flash microcontrollers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI

emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display. Starterkits, eval- and trial-versions are available.

embOS
Real Time Operating System

embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources. The profiling PC tool embOSView is included.

emFile
File system

emFile is an embedded file system with FAT12, FAT16 and FAT32 support. emFile has been optimized for minimum memory consumption in RAM and ROM while maintaining high speed. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

USB-Stack
USB device stack

A USB stack designed to work on any embedded system with a USB client controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer

Flash Programming tool primarily for microcontrollers.

J-Link
JTAG emulator for ARM cores

USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace

USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software

Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Introduction

This guide describes how to use ***embOS*** Real Time Operating System for the Cortex A8 series of microcontrollers using CodeSourcery.

How to use this manual

This manual describes all CPU and compiler specifics of ***embOS*** using Cortex A8 based controllers with CodeSourcery. 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 Cortex A8 and CodeSourcery. 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 detailed information about functionality and fine-tuning of ***embOS*** for the Cortex A8 based controllers using CodeSourcery.

Chapter 2

Using embOS with CodeSourcery

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 CodeSourcery to develop your application, no further installation steps are required. You will find a prepared sample start workspace, which you should use and modify to write your application. So follow the instructions of the next heading *First steps* on page 13.

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

If for some reason you will work with a specific project manager, you should:
Copy either all or only the library-file that you need to your work-directory. Also copy the entire CPU specific subdirectory 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 any project manager or debugger, it may be used in any environment supporting GNU Tools for Cortex A8 without any problem.

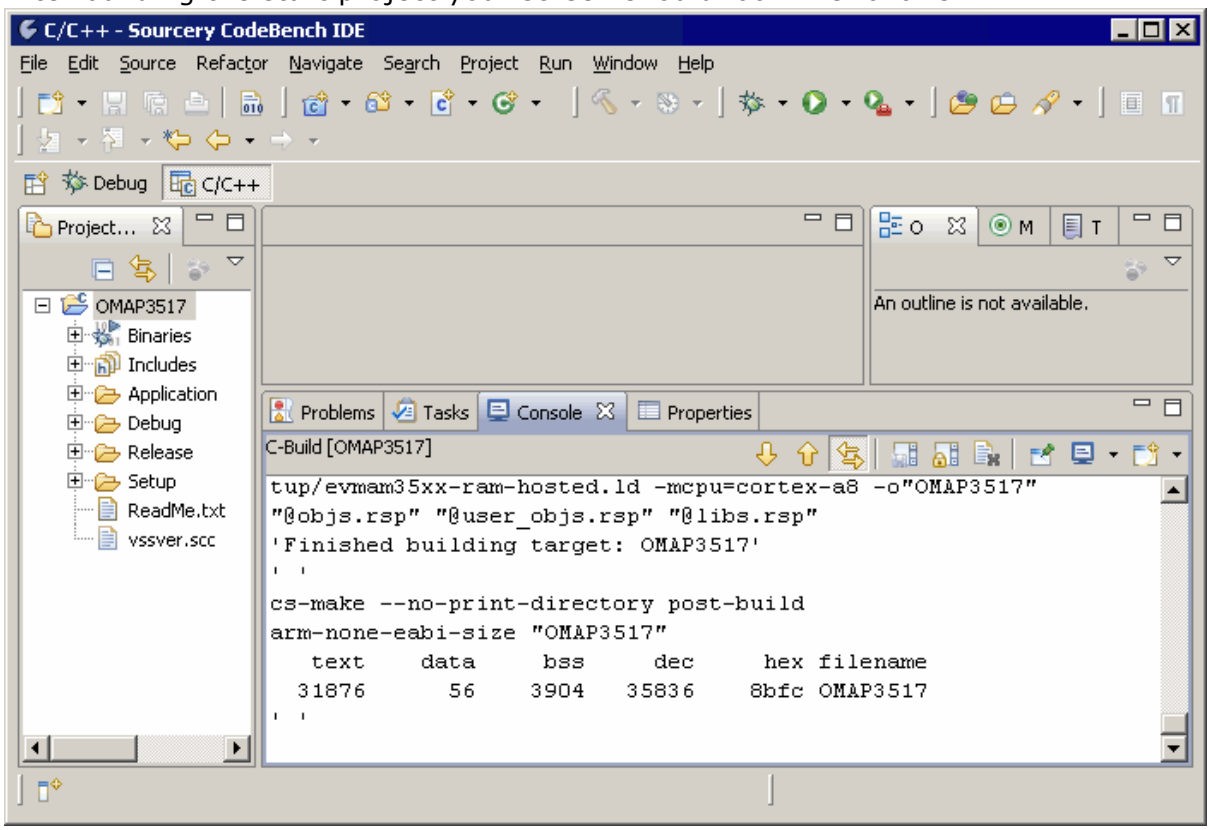
2.2 First steps

After installation of ***embOS*** (See "Installation" on page 12.) you are able to create your first multitasking application. You 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.

To get your first multitasking 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
- Start CodeSourcery and set the workspace e.g. to the "Start\BoardSupport\TI\" folder.

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



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

2.3 The sample application Start2Tasks.c

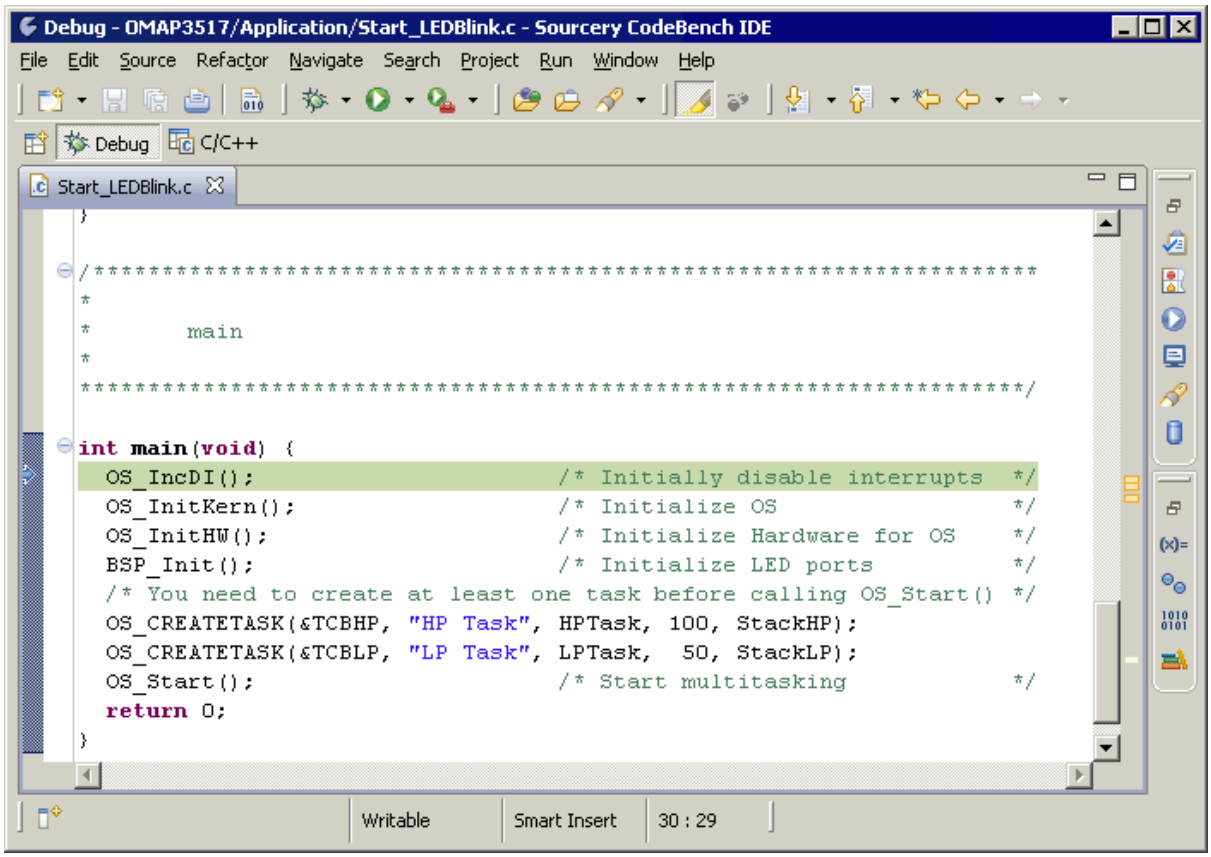
The following is a printout of the sample application `Start2Tasks.c`. It is a good startingpoint for your application. (Please note that the file actually shipped with your port of ***embOS*** may look slightly different from this one)
What happens is easy to see:
After initialization of ***embOS***; 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.

```
-----------------------------------------------------
File    : Start2Tasks.c
Purpose : Skeleton program for OS
--------------END-OF-HEADER----------------------------
*/

#include "RTOS.H"

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

void HPTask(void) {
  while (1) {
    OS_Delay (10);
  }
}

void LPTask(void) {
  while (1) {
    OS_Delay (50);
  }
}

/*****************************************************
*
*       main
*
******************************************************/
int 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(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking            */
  return 0;
}
```

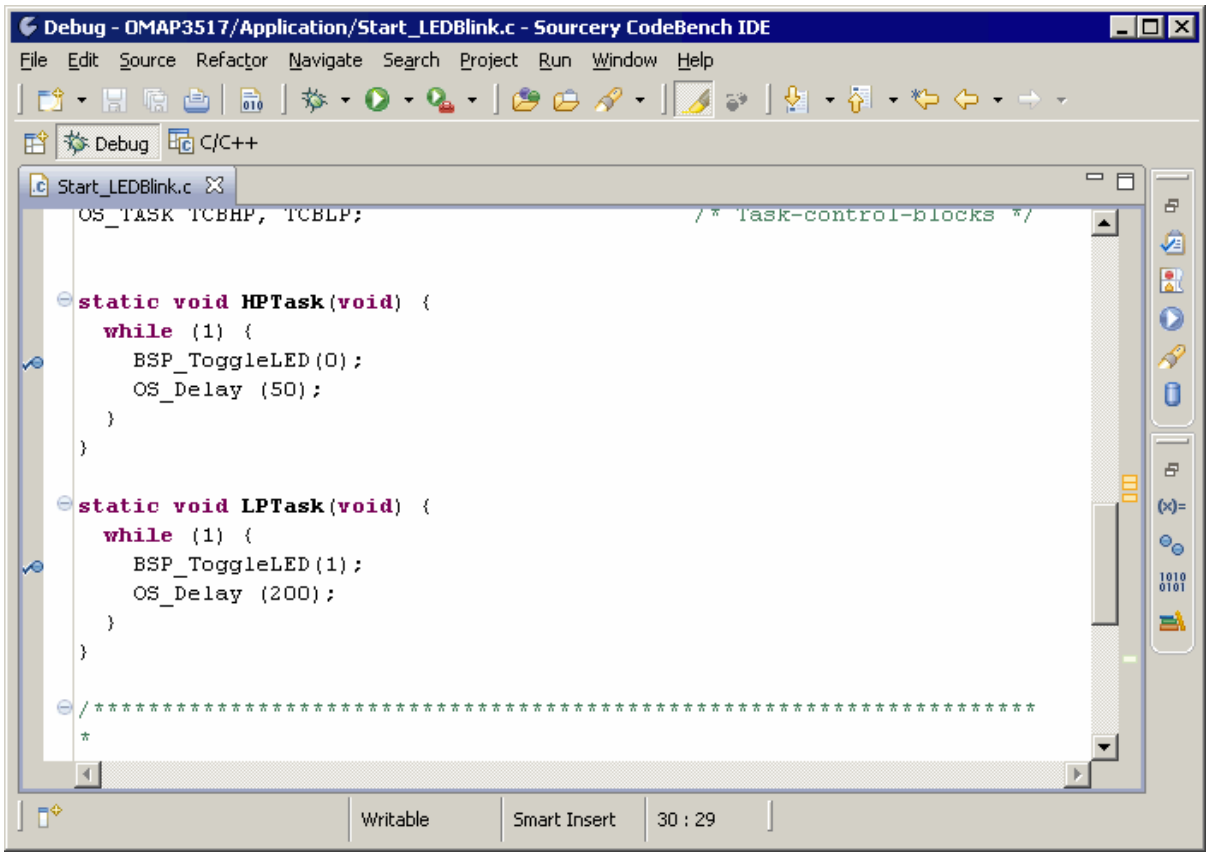

2.4 Stepping through the sample application

embOS comes with debugger settings but CodeSourcery cannot use project relative paths there. Please modify there the paths for "Config" and "Settings file".
Config:
embOS_CA8_CS\CPU\Start\BoardSupport\TI\OMAP3517\Setup/evmam35xx.xml
Settings file:
embOS_CA8_CS\CPU\Start\BoardSupport\TI\OMAP3517\Setup\AM3517.ini

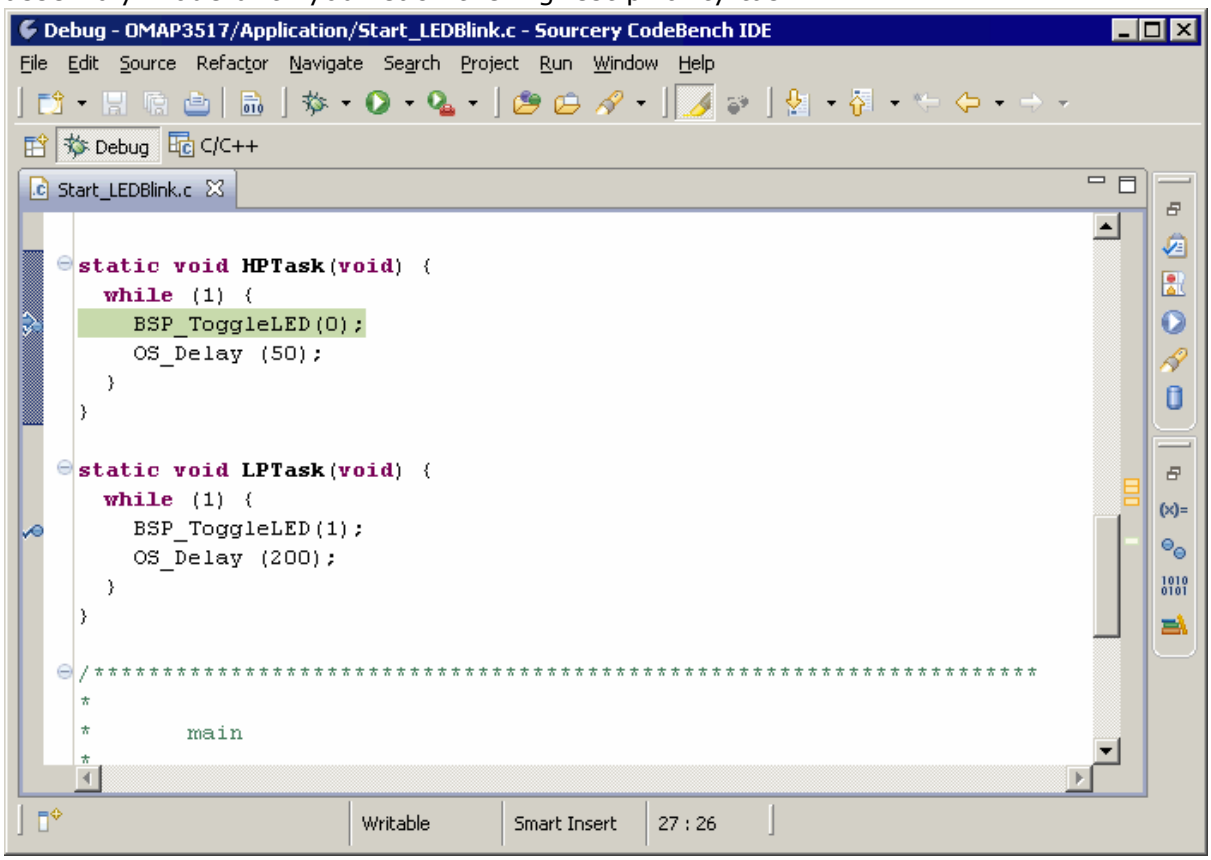
You can start the CodeSourcery debugger with shortcut F11.
In some debuggers, you may look at the startup code and have to set a breakpoint at main. Now you can step through the program.
`OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the ***embOS*** library; you can therefore 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_*.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.



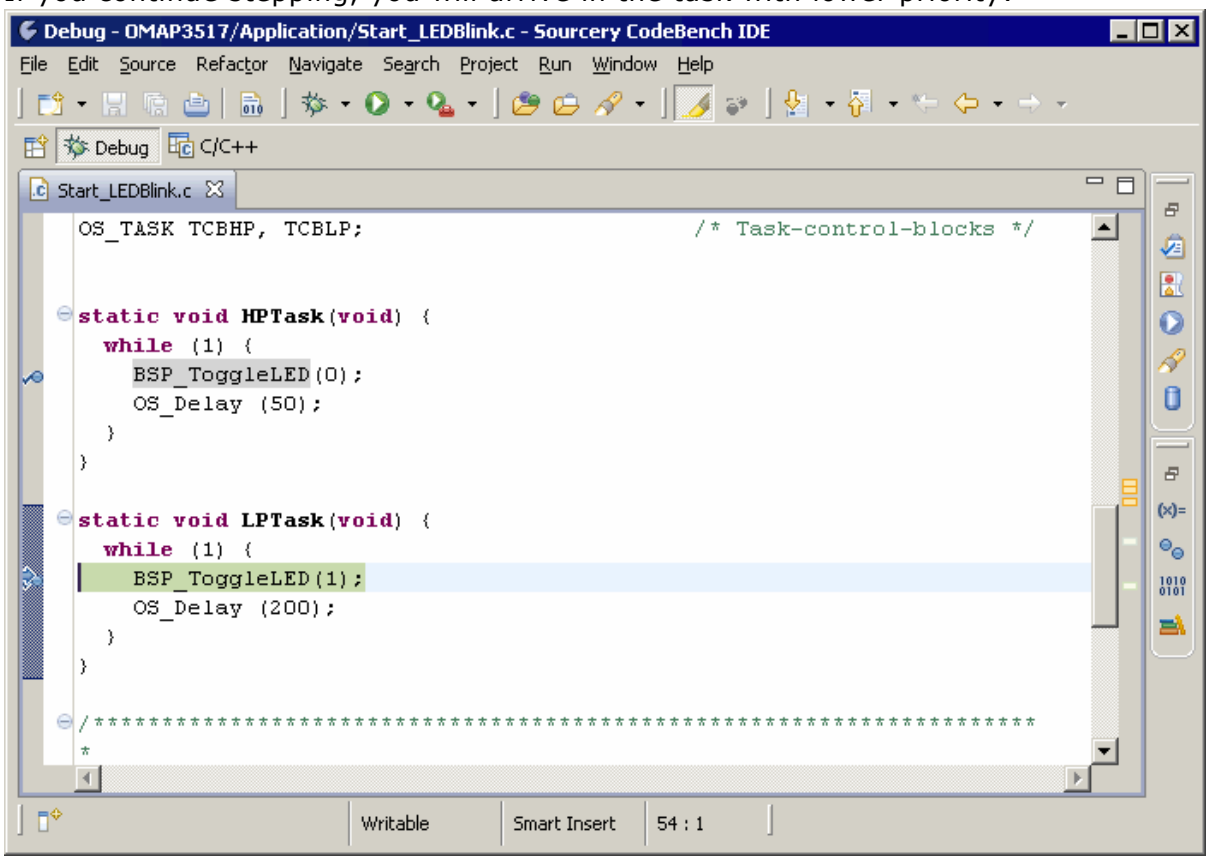
Before you step into `OS_Start()`, you should set two break points in the two tasks as shown below.



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.

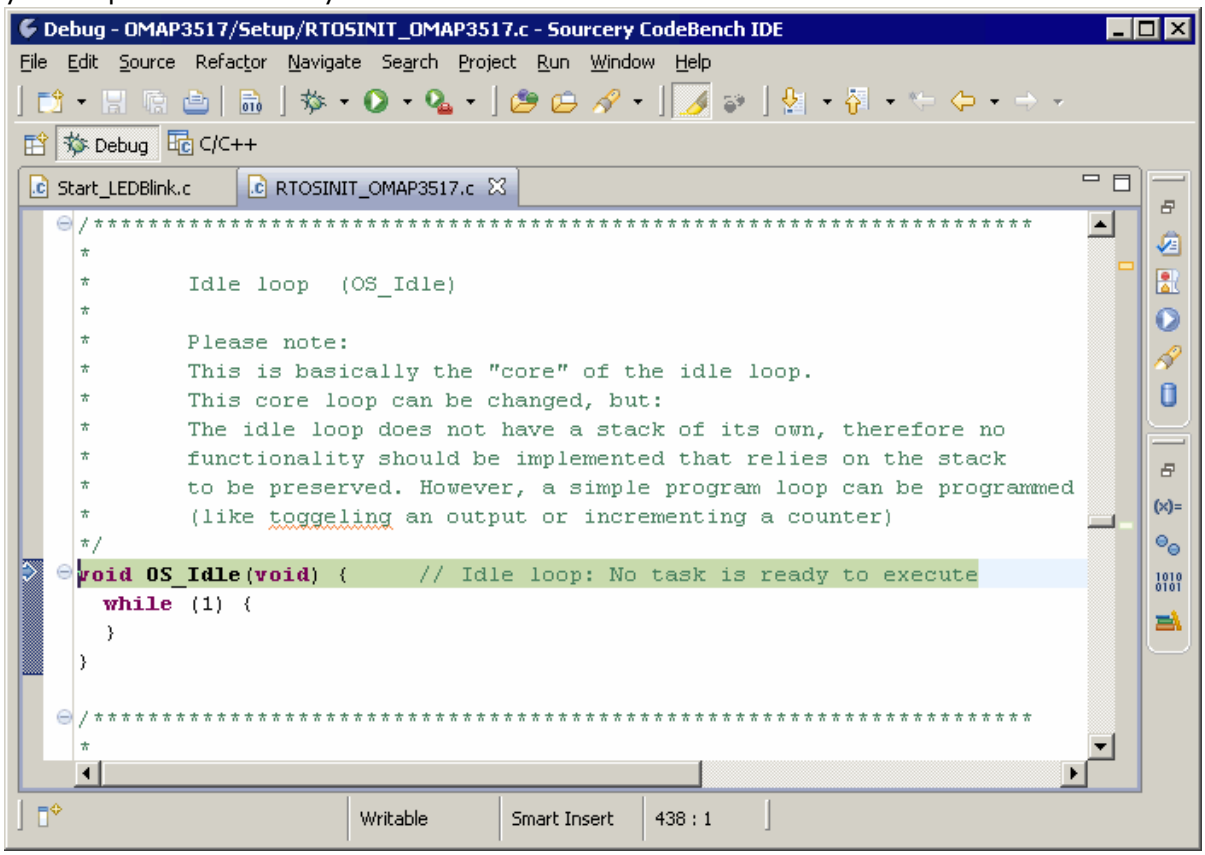


If you continue stepping, you will arrive in the task with lower priority:



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*.c`. You may also set a breakpoint there before you step 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. If you inspect the system variable `OS_Global.Time`, you can see how much time has expired in the target system.
However, when using a simulator, `OS_Global.Time` will not increment, because no timer interrupt is generated. As a result, the program will stick in the idle loop instead of stopping in one of the tasks again.

Chapter 3

Cortex A8 specifics

3.1 CPU modes

embOS supports nearly all memory and code model combinations that GNU Cortex A8 C-Compiler supports.

3.2 Available libraries

embOS for CodeSourcery compiler comes with 28 different libraries, one for each CPU mode / CPU core / endian mode and library type combination.
The libraries are named as follows:

libosca8<m><e><LibMode>.a

Parameter	Meaning	Values
m	CPU mode	A: ARM mode
		T: Thumb2 mode
e	Endian mode	L: Little
		B: Big
LibMode	Library mode	XR: Extreme Release
		R: Release
		S: Stack check
		D: Debug
		SP: Stack check + profiling
		DP: Debug + profiling
		DT: Debug + trace

Example:

`libosca8ALR.a` is the library for a project using ARM mode, little endian mode and release build library type.

Chapter 4

Stacks

4.1 Task stack for Cortex A8

All ***embOS*** tasks execute in system mode. Every ***embOS*** task has its own individual stack which can be located in any memory area. The required stacksize for a task is the sum of the stack-size used by all functions for local variables and parameter passing, 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 Cortex A8, this minimum basic task stack size is about 68 bytes.

4.2 System stack for Cortex A8

The ***embOS*** system executes in supervisor mode. The minimum system stack size required by ***embOS*** is about 136 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 systemstack, the actual stack requirements depend on the application.
The size of the system stack can be changed by modifying "SVC_STACK_SIZE" in your `*.ld` linker script file.

4.3 Interrupt stack for Cortex A8

If a normal hardware exception occurs, the Cortex A8 core switches to IRQ mode,which uses a separate stack pointer. To enable support for nested interrupts,execution of the ISR itself in a different CPU mode than IRQ mode is necessary. ***embOS*** switches to supervisor mode after saving scratch registers, LR_irq and SPSR_irq onto the IRQ stack.
As a result, only registers mentioned above are saved onto the IRQ stack. Forthe interrupt routine itself, the supervisor stack is used. The size of the interrupt stack can be changed by modifying "IRQ_STACK_SIZE" in your `*.ld` linker script file.
Every interrupt requires 28bytes on the interrupt stack.
The maximum interrupt stack size required by the application can be calculated as the Maximum interrupt nesting level * 28 bytes. For task switching from within an interrupt handler, it is required, that the end address of the interrupt stack is aligned to an 8 byte boundary. This alignment is forced during stack pointer initialization in the startup routine. Therefore, an additional margin of about 8 bytes should be added to the calculated maximum interrupt stack size. For standard applications, we recommend at least 92 to 128 bytes of IRQ stack.

4.4 Stack specifics of the Cortex A8 family

Interrupts require space on the supervisor and interrupt stack. The interrupt stack is used to store contents of scratch registers, the ISR itself uses supervisor stack. The Supervisor stack is also used during startup, main(),***embOS*** internal functions and software timers.
All other stacks are not initialized and not used by ***embOS***. If required by the application, the startup function and linker command files have to be modifiedto initialize the stacks.

Chapter 5

Stack Checking

5.1 GCC stack checking

Most applications written in C use a buffer, which is a memory block that holds several instances of the same data type, normally character arrays, on the stack to temporarily hold the intermediate results of string operations. A stack-smashing attack overflows such a buffer by providing a longer string than the actual size of the buffer. This causes the destruction of the contents beyond the buffer, where such contents may include the return address of the caller function and function pointers.

The GCC approach is based on a protection method that automatically inserts protection code into an application at compilation time. It places a random canary between any stack allocated character buffers and the return pointer. It then validates that the canary has not been dirtied by an overflowed buffer before the function returns. It can also reorder local variables to protect local pointers from being overwritten in a buffer overflow.

The GCC Compiler enables stack checking with the options "-fstack-protector" and "-fstack-protector-all". The “-fstack-protector” option only protects functions with character arrays while the “-fstack-protector-all” option protects all functions.

You can use GCC stack checking with embOS for Cortex A8 CPUs without any restrictions.

Please be aware that GCC stack checking needs additional space on the task stacks.

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
- The CPU switches to the Interrupt stack
- The CPU saves PC and flags in registers `LR_irq` and `SPSR_irq`
- The CPU jumps to the vector address `0x18`
- ***embOS*** IRQ_Handler: save scratch registers
- ***embOS*** IRQ_Handler: save `LR_irq` and `SPSR_irq`
- ***embOS*** IRQ_Handler: switch to supervisor mode
- ***embOS*** IRQ_Handler: execute OS_irq_handler (defined in `RTOSInit_*.c`)
- ***embOS*** IRQ_Handler: check for interrupt source and execute timer interrupt, serial communication or user ISR.
- ***embOS*** IRQ_Handler: switch to IRQ mode
- ***embOS*** IRQ_Handler: restore `LR_irq` and `SPSR_irq`
- ***embOS*** IRQ_Handler: pop scratch registers
- Return from interrupt.

When using an Cortex A8 derivate with vectored interrupt controller, please ensure that IRQ_Handler is called from every interrupt. The interrupt vector itself may then be examined by the "C"-level interrupt handler in `RTOSInit_*.c`.

6.2 Defining interrupt handlers in "C"

Interrupt handlers called from ***embOS*** interrupt handler in `RTOSInit*.c` are just normal "C"-functions which do not take parameters and do not return any value.
The default C interrupt handler `OS_irq_handler()` in `RTOSInit*.c` first calls `OS_Enterinterrupt()` or `OS_EnterNestableInterrupt()` to inform ***embOS*** that interrupt code is running. Then this handler examines the source of interrupt and calls the related interrupt handler function.
Finally the default interrupt handler `OS_irq_handler()` in `RTOSInit*.c` calls `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()` and returns to the primary interrupt handler `OS_IRQ_SERVICE()`.
Depending on the interrupting source, it may be required to reset the interrupt pending condition of the related peripherals.

Example of a "simple" interrupt-routine

```
void _OS_Systick(void) {
  TISR = 0x01;
  TCRR = 0;
  OS_HandleTick();
}
```


6.3 Interrupt handling with vectored interrupt controller

For Cortex A8 derivates with built in vectored interrupt controller delivers additional functions to install and setup interrupt handler functions.
When using an Cortex A8 derivate with vectored interrupt controller, please ensure that `IRQ_Handler()` is called from every interrupt. This is default when startup code and hardware initialization delivered with ***embOS*** is used.
The interrupt vector itself will then be examined by the "C"-level interrupt handler `OS_irq_handler()` in `RTOSInit*.c`.
You should not program the interrupt controller for IRQ handling directly. You should use the functions delivered with embOS.

The reaction to an interrupt with vectored interrupt controller is as follows:

- ***embOS*** `IRQ_Handler()` is called by CPU or interrupt controller.
- `IRQ_Handler()` saves registers and switches to supervisor mode.
- `IRQ_Handler()` calls `OS_irq_handler()` (in `RTOSInit*.c`).
- `OS_irq_handler()` examines the interrupting source by reading the interrupt vector from the interrupt controller.
- `OS_irq_handler()` informs ***embOS*** that interrupt code is running by a call of `OS_EnterNestableInterrupt()` which re-enables interrupts.
- `OS_irq_handler()` calls the interrupt handler function which is addressed by the interrupt vector.
- `OS_irq_handler()` resets the interrupt controller to re-enable acceptance of new interrupts.
- `OS_irq_handler()` calls `OS_LeaveNestableInterrupt()` which disables interrupts and informs ***embOS*** that interrupt handling finished.
- `OS_irq_handler()` returns to `IRQ_Handler()`.
- `IRQ_Handler()` restores registers and performs a return from interrupt.

Please note, that different Cortex A8 CPUs may have different versions of vectored interrupt controller hardware and usage of *embOS* supplied functions varies depending on the type of interrupt controller. Please refer to the samples delivered with *embOS* which are used in the CPU specific RTOSInit module.

To handle interrupts with vectored interrupt controller, ***embOS*** offers the following functions:

6.3.1 OS_ARM_InstallISRHandler(): Install an interrupt handler

Description

`OS_ARM_InstallISRHandler()` is used to install a specific interrupt vector when Cortex A8 CPUs with vectored interrupt controller are used.

Prototype

```
OS_ISR_HANDLER* OS_ARM_InstallISRHandler (int             ISRIndex,
                                         OS_ISR_HANDLER* pISRHandler);
```

Parameter	Description
ISRIndex	Index of the interrupt source, normally the interrupt vector number.
pISRHandler	Address of the interrupt handler function.

Table 6.1: OS_ARM_InstallISRHandler() parameter list

Return value

`OS_ISR_HANDLER*`: the address of the previous installed interrupt function, which was installed at the addressed vector number before.

Additional information

This function just installs the interrupt vector but does not modify the priority and does not automatically enable the interrupt.

6.3.2 OS_ARM_EnableISR(): Enable specific interrupt

Description

`OS_ARM_EnableISR()` is used to enable interrupt acceptance of a specific interrupt source in a vectored interrupt controller.

Prototype

```
void OS_ARM_EnableISR(int ISRIndex)
```

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be enabled.

Table 6.2: OS_ARM_EnableISR() parameter list

Additional information

This function just enables the interrupt inside the interrupt controller. It does not enable the interrupt of any peripherals. This has to be done elsewhere.

For Cortex A8 CPUs with VIC type interrupt controller, this function just enables the interrupt vector itself. To enable the hardware assigned to that vector, you have to enable the hardware interrupt enable switch also.

6.3.3 OS_ARM_DisableISR(): Disable specific interrupt

Description

`OS_ARM_DisableISR()` is used to disable interrupt acceptance of a specific interrupt source in a vectored interrupt controller which is not of the VIC type.

Prototype

```
void OS_ARM_DisableISR(int ISRIndex);
```

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be disabled.

Table 6.3: OS_ARM_DisableISR() parameter list

Additional information

This function just disables the interrupt controller. It does not disable the interrupt of any peripherals. This has to be done elsewhere.

When using an Cortex A8 CPU with built in interrupt controller of VIC type, please use `OS_ARM_DisableISRSource()` to disable a specific interrupt.

6.3.4 OS_ARM_ISRSetPrio(): Set priority of specific interrupt

Description

`OS_ARM_ISRSetPrio()` is used to set or modify the priority of a specific interrupt source by programming the interrupt controller.

Prototype

```
int OS_ARM_ISRSetPrio(int ISRIndex,
                      int Prio);
```

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be modified.
`Prio`	The priority which should be set for the specific interrupt.

Table 6.4: OS_ARM_ISRSetPrio() parameter list

Return value

Previous priority which was assigned before the call of `OS_ARM_ISRSetPrio()`.

Additional information

This function sets the priority of an interrupt channel by programming the interrupt controller. Please refer to CPU specific manuals about allowed priority levels.

This function can not be used to modify the interrupt priority for interrupt controllers of the VIC type. The interrupt priority with VIC type controllers depends on the interrupt vector number and can not be changed.

6.4 Interrupt stack switching

Since Cortex A8 core based controllers have a separate stack pointer for interrupts, there is no need for explicit 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. The Cortex A8 interrupt stack is used for primary interrupt handler.

6.5 Fast interrupt FIQ

FIQ interrupt can not be used with ***embOS*** functions, it is reserved for high speed user functions.
`FIQ` is never disabled by ***embOS***.
Never call any ***embOS*** function from an `FIQ` handler.
Do not assign any ***embOS*** interrupt handler to `FIQ`.

When you decide to use `FIQ`, please ensure that `FIQ` stack is initialized during startup and an interrupt vector for `FIQ` handling is included in your application.

Chapter 7

MMU and cache support

7.1 MMU and cache support with embOS

embOS comes with functions to support the MMU and cache of Cortex A8 CPUs which allow virtual-to-physical address mapping with sections of one MByte and cache control. The MMU requires a translation table which can be located in any data area, RAM or ROM, but has to be aligned at a 16Kbyte boundary.

The alignment may be forced by a #pragma or by the linker file. A translation table in RAM has to be set up during run time. embOS delivers API functions to set up this table. Assembly language programming is not required.

7.2 MMU and cache handling for Cortex A8 CPUs

Cortex A8 CPUs with MMU and cache have separate data and instruction caches. embOS delivers the following functions to setup and handle the MMU and caches.

Function	Description
OS_ARM_MMU_InitTT()	Initialize the MMU translation table.
OS_ARM_MMU_AddTTEntries()	Add address entries to the table.
OS_ARM_MMU_Enable()	Enable the MMU.
OS_ARM_ICACHE_Enable()	Enable the instruction cache.
OS_ARM_DCACHE_Enable()	Enable the data cache.
OS_ARM_DCACHE_CleanRange()	Clean memory range in data cache.
OS_ARM_DCACHE_InvalidateRange()	Invalidate memory range in data cache.

Table 7.1: MMU and cache handling for Cortex A8 CPUs

7.2.1 OS_ARM_MMU_InitTT()

Description

`OS_ARM_MMU_InitTT()` is used to initialize an MMU translation table which is located in RAM. The table is filled with zero, thus all entries are marked invalid initially.

Prototype

```
void OS_ARM_MMU_InitTT ( unsigned int * pTranslationTable );
```

Parameter	Description
pTranslationTable	Points to the base address of the translation table.

Table 7.2: OS_ARM_MMU_InitTT() parameter list

Additional Information

This function does not need to be called, if the translation table is located in ROM.

7.2.2 OS_ARM_MMU_AddTTEntries()

Description

`OS_ARM_MMU_AddTTEntries()` is used to add entries to the MMU address translation table. The start address of the virtual address, physical address, area size and cache modes are passed as parameter.

Prototype

```
void OS_ARM_MMU_AddTTEntries ( unsigned int * pTranslationTable,
                               unsigned int   CacheMode,
                               unsigned int   VIndex,
                               unsigned int   PIndex,
                               unsigned int   NumEntries );
```

Parameter	Description
`pTranslationTable`	Points to the base address of the translation table.
`CacheMode`	Specifies the cache operating mode which should be used for the selected area. May be one of the following modes: `OS_ARM_CACHEMODEA7_NC_NB` - non cacheable, non bufferable `OS_ARM_CACHEMODEA7_C_NB` - cacheable, non bufferable `OS_ARM_CACHEMODEA7_NC_B` - non cacheable, bufferable `OS_ARM_CACHEMODEA7_C_B` - cacheable, bufferable
`VIndex`	Virtual address index, which is the start address of the virtual memory address range with MBytes resolution. `VIndex` = (virtual address >> 20)
`PIndex`	Physical address index, which is the start address of the physical memory area range with MBytes resolution. `PIndex` = (physical address >> 20)
`NumEntries`	Specifies the size of the memory area in MBytes.

Table 7.3: OS_ARM_MMU_AddTTEntries() parameter list

Additional Information

This function does not need to be called, if the translation table is located in ROM. The function adds entries for every section of one MegaByte size into the translation table for the specified memory area.

7.2.3 OS_ARM_MMU_Enable()

Description

`OS_ARM_MMU_Enable()` is used to enable the MMU which will then perform the address mapping.

Prototype

```
void OS_ARM_MMU_Enable ( unsigned int * pTranslationTable );
```

Parameter	Description
`pTranslationTable`	Points to the base address of the translation table.

Table 7.4: OS_ARM_MMU_Enable() parameter list

Additional Information

As soon as the function was called, the address translation is active. The MMU table has to be setup before calling `OS_ARM_MMU_Enable()`.

7.2.4 OS_ARM_ICACHE_Enable()

Description

`OS_ARM_ICACHE_Enable()` is used to enable the instruction cache of the CPU.

Prototype

```
void OS_ARM_ICACHE_Enable ( void );
```

Additional Information

As soon as the function was called, the instruction cache is active. It is CPU implementation defined whether the instruction cache works without MMU. Normally, the MMU should be setup before activating instruction cache.

7.2.5 OS_ARM_DCACHE_Enable()

Description

`OS_ARM_DCACHE_Enable()` is used to enable the data cache of the CPU.

Prototype

```
void OS_ARM_DCACHE_Enable ( void );
```

Additional Information

The function must not be called before the MMU translation table was set up correctly and the MMU was enabled. As soon as the function was called, the data cache is active, according to the cache mode settings which are defined in the MMU translation table. It is CPU implementation defined whether the data cache is a write through, a write back, or a write through/write back cache. Most modern CPUs will have implemented a write through/write back cache.

7.2.6 OS_ARM_DCACHE_CleanRange()

Description

`OS_ARM_DCACHE_CleanRange()` is used to clean a range in the data cache memory to ensure that the data is written from the data cache into the memory.

Prototype

```
void OS_ARM_DCACHE_CleanRange ( void *       p,
                                unsigned int NumBytes );
```

Parameter	Description
p	Points to the base address of the memory area that should be updated.
NumBytes	Number of bytes which have to be written from cache to memory.

Table 7.5: OS_ARM_DCACHE_CleanRange() parameter list

Additional Information

Cleaning the data cache is needed, when data should be transferred by a DMA or other BUS master that does not use the data cache. When the CPU writes data into a cacheable area, the data might not be written into the memory immediately. When then a DMA cycle is started to transfer the data from memory to any other location or peripheral, the wrong data will be written.

Before starting a DMA transfer, a call of `OS_ARM_DCACHE_CleanRange()` ensures, that the data is transferred from the data cache into the memory and the write buffers are drained.

The cache is cleaned line by line. Cleaning one cache line takes approximately 10 CPU cycles. As each cache line covers 64 bytes, the total time to invalidate a range may be calculated as:

t = (NumBytes / 64) * (10 [CPU clock cycles] + Memory write time).

The real time depends on the content of the cache. If data in the cache is marked as dirty, the cache line has to be written to memory. The memory write time depends on the memory BUS clock and memory speed. If data has to be written to memory, the required cycles for this memory operation has to be added to the 10 CPU clock cycles for every 64 bytes to be cleaned.

7.2.7 OS_ARM_DCACHE_InvalidateRange()

Description

OS_ARM_DCACHE_InvalidateRange() is used to invalidate a memory area in the data cache. Invalidating means, mark all entries in the specified area as invalid. Invalidation forces re-reading the data from memory into the cache, when the specified area is accessed again.

Prototype

```
void OS_ARM_DCACHE_InvalidateRange ( void *       p,
                                     unsigned int NumBytes );
```

Parameter	Description
SourceIndex	Index of the interrupt channel which should be disabled.

Table 7.6: OS_ARM_DCACHE_InvalidateRange() parameter list

Additional Information

This function is needed, when a DMA or other BUS master is used to transfer data into the main memory and the CPU has to process the data after the transfer.

To ensure, that the CPU processes the updated data from the memory, the cache has to be invalidated. Otherwise the CPU might read invalid data from the cache instead of the memory.

Special care has to be taken, before the data cache is invalidated. Invalidating a data area marks all entries in the data cache as invalid. If the cache contained data which was not written into the memory before, the data gets lost. Unfortunately, only complete cache lines can be invalidated.

Therefore, it is requires, that the base address of the memory area has to be located at a 64 byte boundary and the number of bytes to be invalidated has to be a multiple of 64 bytes.

The debug version of embOS will call OS_Error() with error code OS_ERR_NON_ALIGNED_INVALIDATE, if one of these restrictions is violated.

The cache is invalidated line by line. Invalidating one cache line takes approximately 10 CPU cycles. As each cache line covers 64 bytes, the total time to invalidate a range may be calculated as:

t = (NumBytes / 64) * 10 [CPU clock cycles].

7.3 MMU and cache handling program sample

The MMU und cache handling has to be set up before the data segments are initialized. Otherwise a virtual address mapping would not work. The startup code calls the `__low_level_init()` function before sections are initialized.

It is a good idea to initialize memory access, the MMU table and the cache control during `__low_level_init()`. The following sample is an excerpt from one `__low_level_init()` function which is part of an `RTOSInit.c` file:

```
/*********************************************************************
*
* MMU and cache configuration
*
* The MMU translation table has to be aligned to 16KB boundary
* and has to be located in uninitialized data area
*/
static unsigned int _TranslationTable [0x1000];
__low_level_init(void) {
  //
  // Initialize SDRAM
  //
  _InitSDRAM();
  //
  // Init MMU and caches
  //
  OS_ARM_MMU_InitTT (&_TranslationTable[0]);
  //
  // SDRAM, the first MB remapped to 0 to map vectors to correct address,
  //cacheable, bufferable
  OS_ARM_MMU_AddTTEntries ( &_TranslationTable[0],
                            OS_ARM_CACHEMODEA7_C_B,
                            0x000, 0x200, 0x001);
  // Internal SRAM, original address, NON cachable, NON bufferable
  OS_ARM_MMU_AddTTEntries ( &_TranslationTable[0],
                            OS_ARM_CACHEMODEA7_NC_NB,
                            0x003, 0x003, 0x001);
  OS_ARM_MMU_Enable (&_TranslationTable[0]);
  OS_ARM_ICACHE_Enable();
  OS_ARM_DCACHE_Enable();
  return 1;
}
```

Other samples are included in the CPU specific `RTOSInit*.c` files delivered with embOS.

Chapter 8

STOP / WAIT mode

8.1 Saving power

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 `RTOSIinit_*.c`.

Chapter 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***. Using Cortex A8 with ARM mode, the minimum ROM requirement for the kernel itself is about 2.500 bytes. In the table below, you find the minimum RAM size for ***embOS*** resources. The sizes depend on selected ***embOS*** library mode; the table below is for a release build.

embOS resource	RAM [bytes]
Task control block	32
Resource semaphore	8
Counting semaphore	4
Mailbox	20
Software timer	20

Chapter 10

Files shipped with embOS

10.1 Files included in embOS

Directory	File	Explanation
root	*.pdf	Generic API and target specific documentation
root	embOSView.exe	Utility for runtime analysis, described in generic documentation
root	Release.html	Version control document
Start\BoardSupport*\Application\	*.*	Sample programs to serve as a start
Start\BoardSupport*\Setup\	*.*	CPU specific hardware routines for various CPUs
Start\BoardSupport*\	make	Sample MAKEFILE that can be used with the GNU make tool
Start\BoardSupport*\Setup\	Startup.s	Generic stack/RAM initialization file
Start\Inc\	BSP.h	Include file for BoardSupport packages, to be included in every "C"-file using BSP-functions
Start\Inc\	OS_Config.h	Include file for ***embOS*** library mode configuration, included by RTOS.h
Start\Inc\	RTOS.h	Include file for ***embOS***, to be included in every "C"-file using embOS-functions
Start\Lib\	os*.a	***embOS*** libraries

Any additional files shipped serve as example.

Index

