embOS

Operating System

CPU & Compiler specifics for Altera SoC using ARM DS-5 and GNU compiler

Document: UM01053
Software version 4.04
Revision: 0
Date: November 21, 2014

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

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
E-mail: support@segger.com
Internet: http://www.segger.com

Manual versions

This 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: November 21, 2014

Software	Revision	Date	By	Description
4.04	0	141121	MC	NEON support added.
4.02a	1	141021	MC	Minor corrections.
4.02a	0	141007	MC	Update to latest embOS generic sources 4.02a
3.88h	0	140109	AW	Initial version, based on embOS sources 3.88h

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.
`Sample comment`	Comments in programm 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 2.1: Typographic conventions

USB
j-link
SEGGER
www.segger.com
JTAG

Table of Contents

Chapter 1

Using embOS ARM GNU Altera

This chapter describes how to start with and use embOS ARM GNU Altera with the ARM DS-5 workbench and GNU tool chain.
You should follow these steps to become familiar with embOS.

1.1 Installation

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

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

Assuming that you are using the Eclipse for DS-5 to develop your application, no further installation steps are required. You will find a prepared sample start project which you should use and modify to write your application. So follow the instructions of section “First steps” on page 11.

You should do this even if you do not intend to use the Eclipse based workbench for your application development to become familiar with embOS.

If you will not work with the Eclipse based workbench, you should: Copy the whole folder "Start" of the embOS shipment into your work directory. A makefile is found in the board specific subfolder. The project can be built by using the makefile. It may be required to setup your environment, so that the compiler is found in the path. embOS does in no way rely on the Eclipse based DS-5 workbench, it may be used without the project manager using batch files or a make utility without any problem.

1.2 First steps

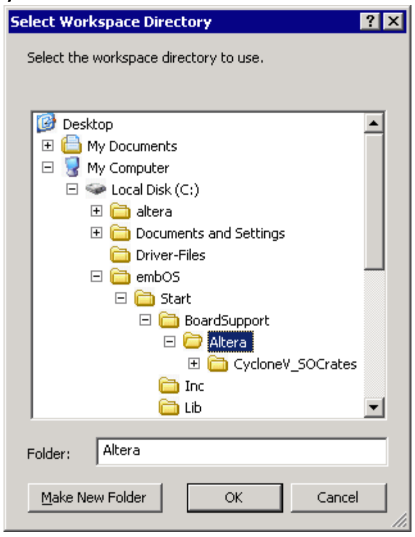
After installation of embOS you are able to create and run your first multitasking application. You received a ready to go sample project and it is a good idea to use this as a starting point for your applications.
Your embOS distribution contains one folder "Start\BoardSupport" which contains the sample project files and every additional file used to build your application.

The following chapters describe a session using the sample project for the Altera Cyclone V SoC on the SoCrates Cyclone V Evaluation Board.
The project is located at "Start\BoardSupport\Altera\CycloneV_SOCrates".

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

- Create a work directory for your application, for example `c:\embOS`.
- 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 the Eclipse for DS-5 workbench and select and create a workspace.
- Import the sample project into the workspace
- Build the start project
- Run the application using the USB Blaster II debugger for downloading and debugging.

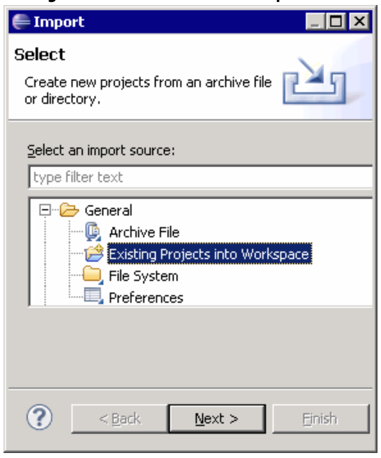
Start the DS-5 workbench and in the workspace launcher click "Browse..." to select the workspace. If the workspace launcher is not shown on startup, select it by menu "File -> Switch Workspace".
Select the workspace directory "Start\BoardSupport\Altera" or any other folder on your hard disk.


Select Workspace Directory

Select the workspace directory to use.

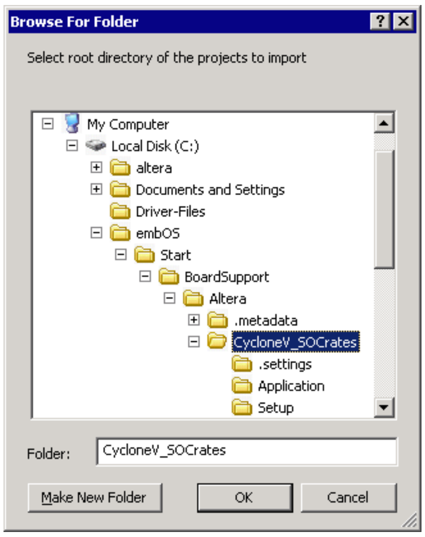
Folder: Altera

The workspace will then be created in the selected folder.
Now import the sample start project from the folder CycloneV_SOCrates.
Choose menu "File -> Import" and in "Import" dialog select "General -> Existing Projects into workspace".



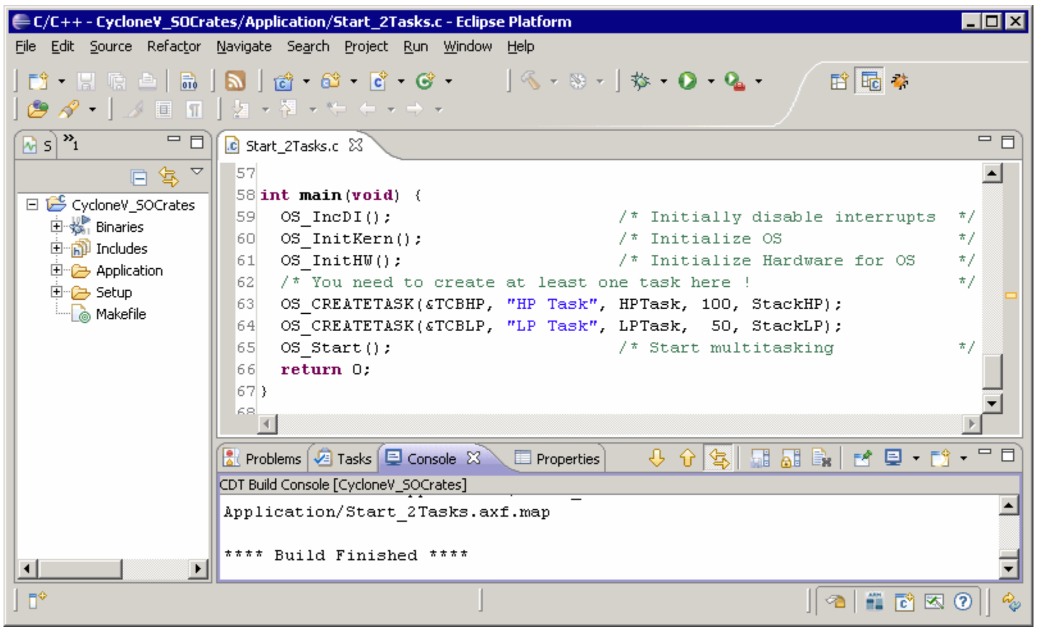
Press "Next", the Import Projects dialog shows up.

Press "Browse..." and select "Start\Boardsupport\Altera\CycloneV_SOCrates" as the root directory for the project to import:



Press "OK".

Refresh the project and build it:


C/C++ - CycloneV_SOCrates/Application/Start_2Tasks.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
CycloneV_SOCrates
Binaries
Includes
Application
Setup
Makefile
Start_2Tasks.c
Problems
Tasks
Console
Properties

```
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;
}
```

CDT Build Console [CycloneV_SOCrates]

```
Application/Start_2Tasks.axf.map

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

1.3 The example application Start_2Tasks.c

The following is a printout of the example application `Start_2Tasks.c`. It is a good starting point for your application. (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.

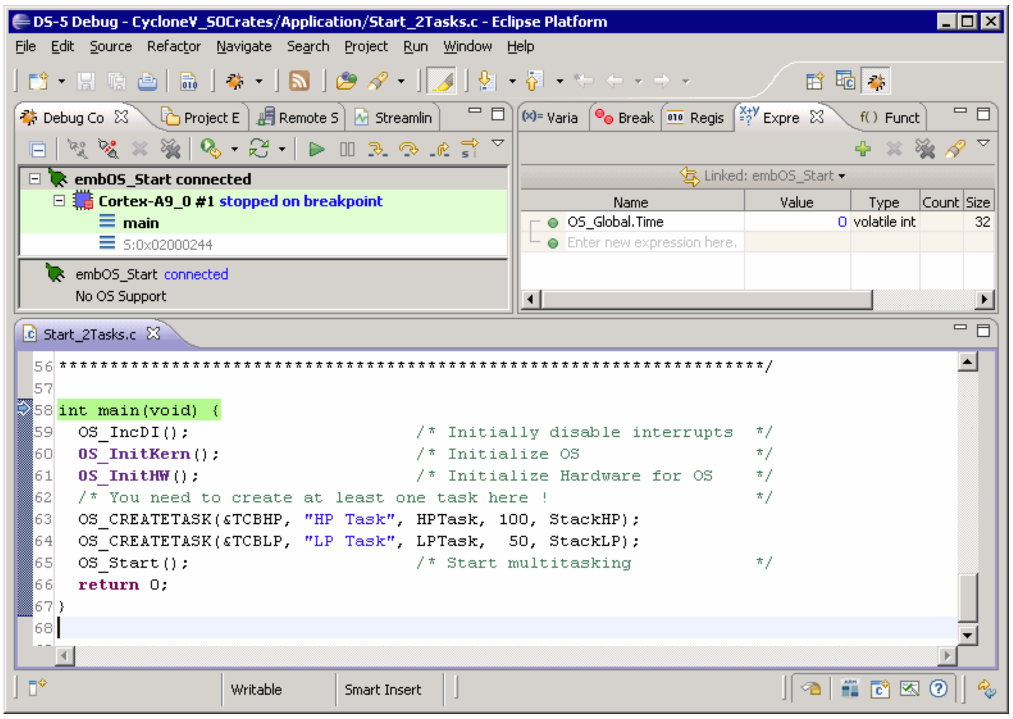
```
/**********************************************************
* SEGGER MICROCONTROLLER SYSTEME GmbH & Co.KG
* Solutions for real time microcontroller applications
**********************************************************
File    : Start_2Tasks.c
Purpose : Skeleton program for OS
--------- END-OF-HEADER ---------------------------------*/

#include "RTOS.h"

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

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

static 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    */
  BSP_Init();                       /* Initialize LED ports          */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                       /* Start multitasking            */
  return 0;
}
```

1.4 Stepping through the sample application

The embOS start project comes with a debug configuration embOS_Start, which can be used to download and debug the start project in the SDRAM of the eval bord. When starting the debugger, you will usually see the `main()` function (very similar to the example screenshot below). With some debug configurations, you may look at the startup code at the program entry point. In this case, you should set a breakpoint at main() and start the execution to reach main().

Now you can step through the program.
`OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the embOS library; you can therefore step into it in disassembly mode only. 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, because it starts multitasking and does not return.

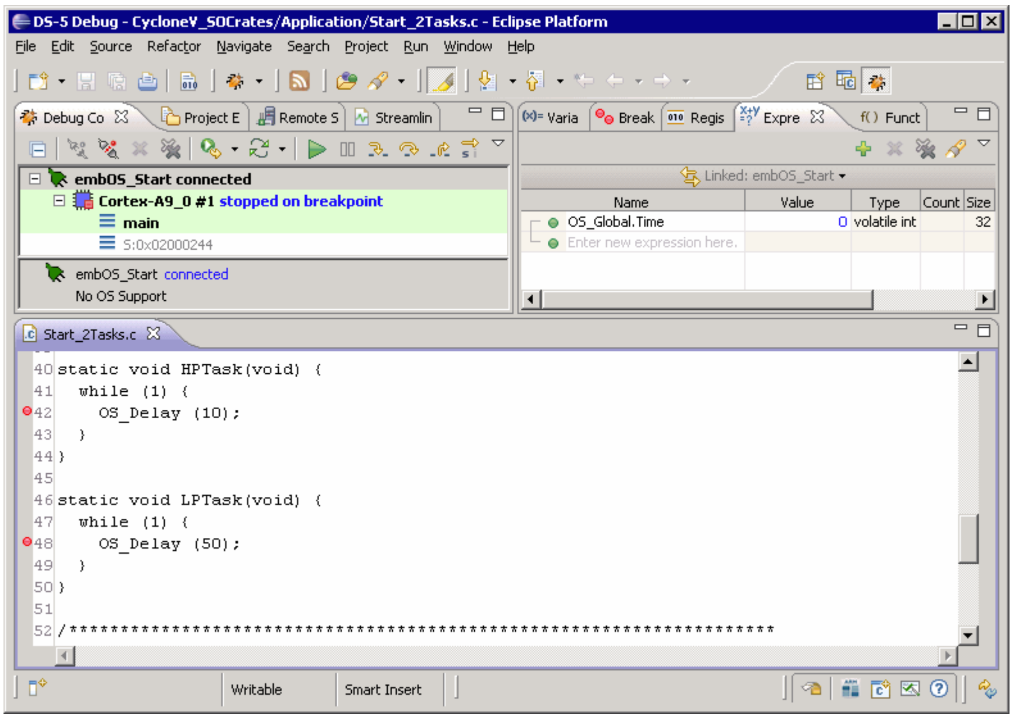

DS-5 Debug - CycloneV_SOCrates/Application/Start_2Tasks.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug Co
Project E
Remote S
Streamlin
Varia
Break
Regis
Expre
Funct
Linked: embOS_Start
Start_2Tasks.c
Writable
Smart Insert

```
embOS_Start connected
  Cortex-A9_0 #1 stopped on breakpoint
    main
    S:0x02000244
embOS_Start connected
No OS Support
```

Name	Value	Type	Count	Size
OS_Global.Time	0	volatile int		32
Enter new expression here.				

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

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;
}
```

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


DS-5 Debug - CycloneV_SOCrates/Application/Start_2Tasks.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug Co
Project E
Remote S
Streamlin
Varia
Break
Regis
Expre
Funct
Start_2Tasks.c
Writable
Smart Insert

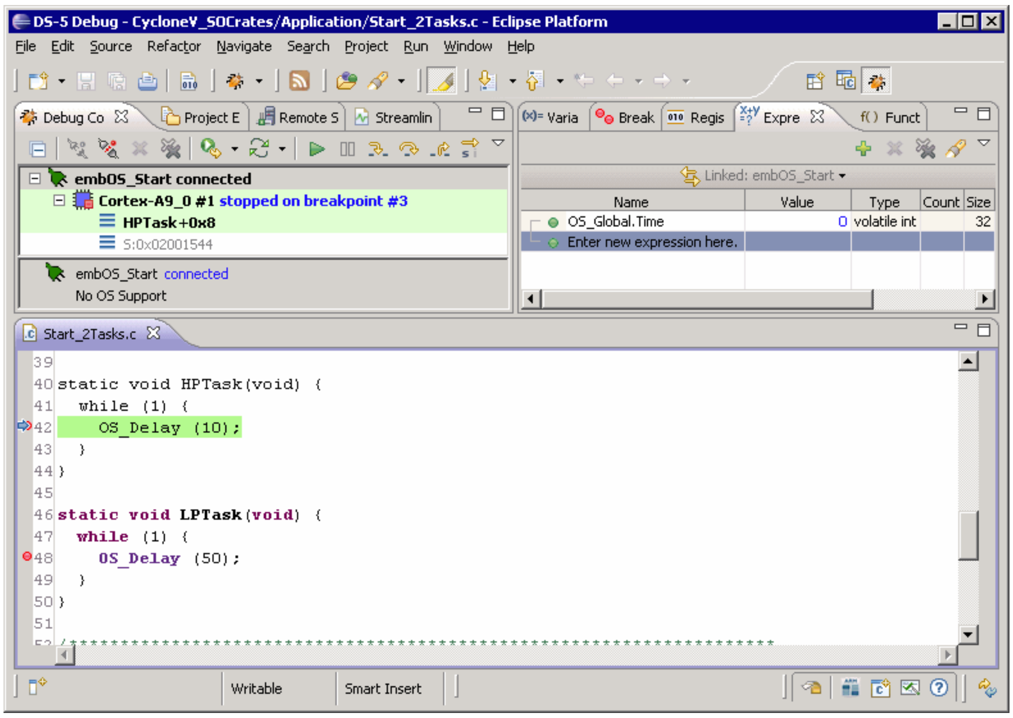
```
embOS_Start connected
  Cortex-A9_0 #1 stopped on breakpoint
    main
    S:0x02000244
embOS_Start connected
No OS Support
```

Linked: embOS_Start

Name	Value	Type	Count	Size
OS_Global.Time	0	volatile int		32
Enter new expression here.				

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

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

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

As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only. Click **Continue (F8)**, step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task.


DS-5 Debug - CycloneV_SOCrates/Application/Start_2Tasks.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug Co
Project E
Remote S
Streamlin
Varia
Break
Regis
Expre
Funct
Start_2Tasks.c
Writable
Smart Insert

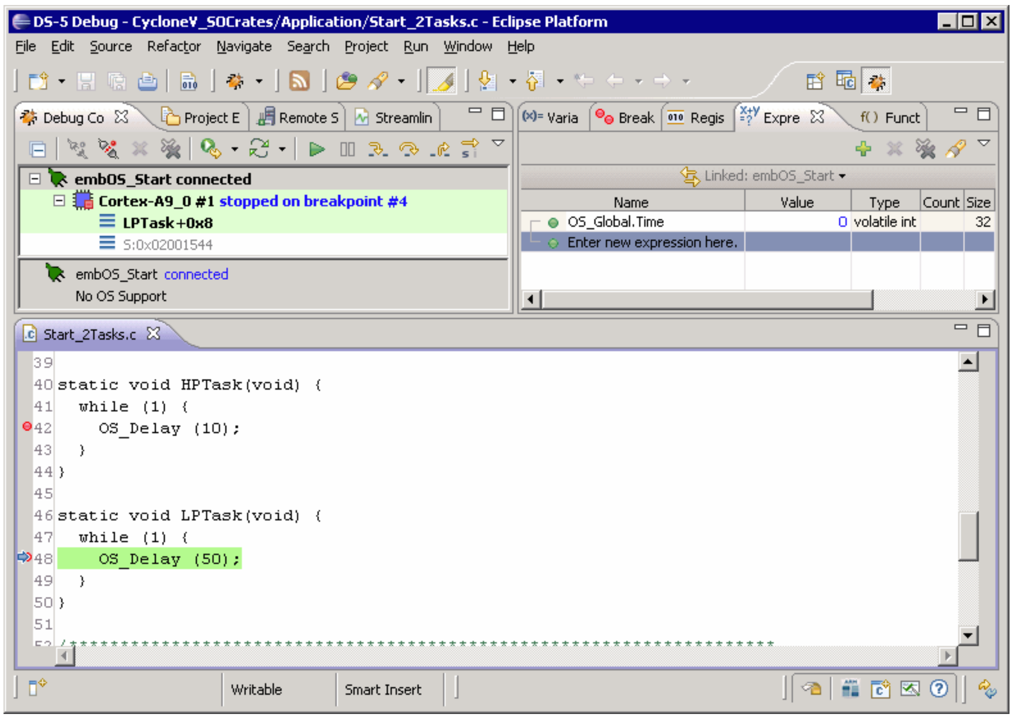
```
embOS_Start connected
  Cortex-A9_0 #1 stopped on breakpoint #3
    HPTask+0x8
    S:0x02001544
embOS_Start connected
No OS Support
```

Linked: embOS_Start

Name	Value	Type	Count	Size
OS_Global.Time	0	volatile int		32
Enter new expression here.				

```

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

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

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

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


DS-5 Debug - CycloneV_SOCrates/Application/Start_2Tasks.c - Eclipse Platform

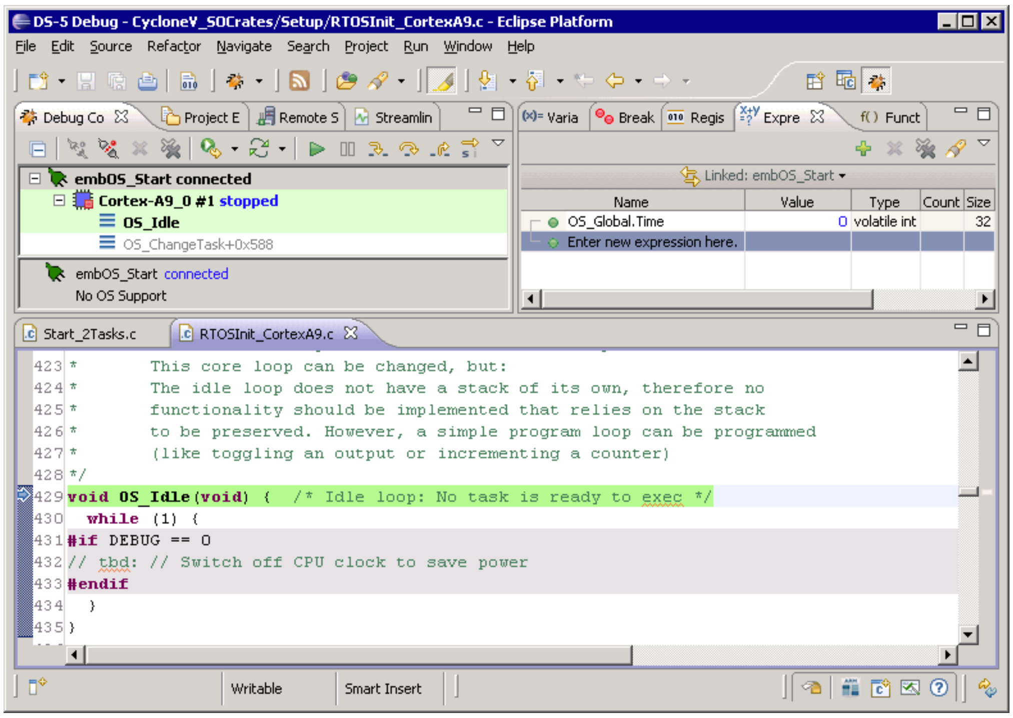
```
embOS_Start connected
  Cortex-A9_0 #1 stopped on breakpoint #4
    LPTask+0x8
    S:0x02001544
embOS_Start connected
No OS Support
```

Name	Value	Type	Count	Size
OS_Global.Time	0	volatile int		32
Enter new expression here.				

Start_2Tasks.c

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

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

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

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`.


DS-5 Debug - CycloneV_SOCrates/Setup/RTOSInit_CortexA9.c - Eclipse Platform

```
embOS_Start connected
  Cortex-A9_0 #1 stopped
    OS_Idle
    OS_ChangeTask+0x588
embOS_Start connected
No OS Support
```

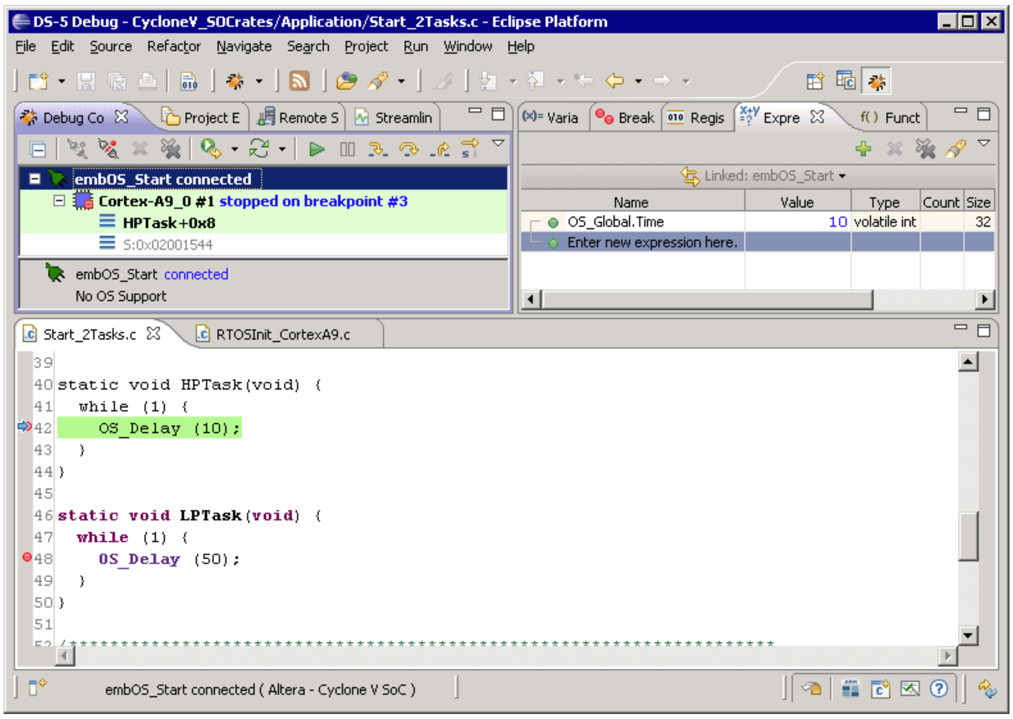
Name	Value	Type	Count	Size
OS_Global.Time	0	volatile int		32
Enter new expression here.				

RTOSInit_CortexA9.c

```
*       This core loop can be changed, but:
*       The idle loop does not have a stack of its own, therefore no
*       functionality should be implemented that relies on the stack
*       to be preserved. However, a simple program loop can be programmed
*       (like toggling an output or incrementing a counter)
*/
void OS_Idle(void) {   /* Idle loop: No task is ready to exec */
  while (1) {
#if DEBUG == 0
// tbd: // Switch off CPU clock to save power
#endif
  }
}
```

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 the embOS timer variable OS_Global.Time, shown in the Watch window, `HPTask` continues operation after expiration of the 10 ms delay.


DS-5 Debug - CycloneV_SOCrates/Application/Start_2Tasks.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug Co
Project E
Remote S
Streamlin
Varia
Break
Regis
Expre
Funct
Linked: embOS_Start
Start_2Tasks.c
RTOSInit_CortexA9.c

embOS_Start connected
Cortex-A9_0 #1 stopped on breakpoint #3
HPTask+0x8
S:0x02001544

embOS_Start connected
No OS Support

Name	Value	Type	Count	Size
OS_Global.Time	10	volatile int		32
Enter new expression here.				

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

static void LPTask(void) {
  while (1) {
    OS_Delay (50);
  }
}
/*********************************************************************
```

embOS_Start connected (Altera - Cyclone V SoC)

Chapter 2

Build your own application

This chapter provides all information to setup your own embOS project.

2.1 Introduction

To build your own application, you should always start with the supplied sample project. Therefore, select and build an embOS workspace as described in First steps on page 9 and modify the project to fit your needs. Using a sample project as starting point has the advantage that all necessary files are included and all settings for the project are already done.
You may add your own code and sources by just copying the files in the root or application folder of the project and modifiy the make file.

2.2 Required files for embOS for Altera SoC and DS-5

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 subfolder `Inc\`.
  This header file declares all embOS API functions and data types and has to be included in any source file using embOS functions. The include path in the make file has to be setup to find the RTOS.h file.
- **RTOSInit_*.c** from the target specific folder. In our sample project, RTOSOInit is located in
  Start\BoardSupport\Altera\CycloneV_SOCrates\Setup\
  RTOSInit contains hardware-dependent initialization code for embOS. It initializes the system timer, the interrupt controller, the MMU, the timer interrupt and optional communication for embOSView via UART.
- One embOS library from the subfolder `Lib\`.
- OS_Error.c from target specific subfolder
  BoardSupport\Altera\CyloneV_SOCrates\Setup\.
  The error handler is used if any library other than a Release build library is used in your project.
- **startup.S** from the **BoardSupport\Altera\CyloneV_SOCrates\Setup\** folder. It contains the startup code wich is required to perform a low level CPU configuration, initializes the MMU and caches by a call of the `__low_level_init()` function from RTOSInit and sets up initialized and zero initialized variables.
- Preloader, second level loader and debug scripts are required to get the current version of embOS ARM GNU Altera running on the target using the debugger. All required files are located in the Setup folder.

When you decide to write your own startup code, ensure, that at least the SVC and IRQ stack pointeres are initialized using the stack addresses described later on.
Ensure, `__low_level_init()` is called before the variables are initialized. Ensure that non-initialized variables are initialized with zero, according to C standard. This is required for some embOS internal variables.
Finally ensure, that `main()` is called with the CPU running in supervisor or system mode. Interrupts should be disabled when entering `main()`.
Your `main()` function has to initialize embOS by a call of `OS_InitKern()` and `OS_InitHW()` prior any other embOS embOS functions except OS_IncDI() are called.
You should then modify or replace the sample `Start_2Task.c` source file in the subfolder `Application\`, or modify the makefile to compile and link another application.

2.3 Change library mode

For your application you might want to choose another 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 select or replace the embOS library in your project or target:

- Include the wished library by modification of the makefile.
- Check and set the appropriate `OS_LIBMODE_*` define as compiler flag in the makefile.

Chapter 3

CPU specifics

3.1 CPU modes

embOS for ARM cores with GNU compiler supports all memory and code model combinations that the GNU compiler for ARM supports.
As this version of embOS for ARM is dedicated to Altera SoC, which is a Cortex-A9 multi core, the shipment comes with libaries for ARM9 CPUs only, which can be used with Cortex-A9.
The source version of embOS allows generation of all supported libraries.

3.2 Available embOS libraries

embOS for ARM GNU Altera comes with 7 different libraries, compiled for ARMv5 architecture. These can be used with Cortex A9 cores. Cortex A specific code for MMU and VFP support is included in the libraries.
The library names follow the naming conventions described below.

3.2.1 Naming conventions for prebuilt embOS libraries

embOS ARM GNU Altera is shipped with different prebuilt libraries with different combinations of the following features:

- Instruction set architecture - `Arch`
- CPU mode - `CpuMode`
- Byte order - `ByteOrder`
- Library mode - `LibMode`

The libraries are named as follows:

`libos<CpuMode><Arch><ByteOrder><Interwork><Libmode>.a`

Parameter	Meaning	Values
`CpuMode`	Specifies the CPU mode.	A: ARM mode T: Thumb mode *
`Arch`	Specifies the CPU variant	4: ARM 7 (architecture 4) * 5: ARM 9 (architecture 5)
`ByteOrder`	Specifies target endianess.	B: Big endian * L: Little endian
`Interwork`	Specifies if interwork option is enabled.	N: No interworking.
`LibMode`	Specifies the library mode	XR: Extreme Release
		R: Release
		S: Stack check
		SP: Stack check + Profiling
		D: Debug
		DP: Debug + Profiling
		DT: Debug + Profiling + Trace

* = Not delivered with embOS ARM GNU Altera, but can be generated with the source version of embOS.

Example

libosA5LNDP.a is the library for a project using an ARM9 or Cortex A9 core, ARM mode, little endian mode, no interworking, with debug and profiling support.

Chapter 4

Compiler specifics

4.1 Standard system libraries

embOS for ARM GNU Altera may be used with standard GNU system libraries for most of all projects without any modification.

Heap management and file operation functions of standard system libraries are not reentrant and require a special initialization or additional modules when used with **embOS**, if non thread safe functions are used from different tasks.

Alternatively, for heap management, **embOS** delivers its own thread safe functions which may be used. These functions are described in the **embOS** generic manual.

4.2 Reentrancy, thread local storage

The GCC newlib supports usage of thread-local storage located in a _reent structure as local variable for every task.
Several library objects and functions need local variables which have to be unique to a thread. Thread-local storage will be required when these functions are called from multiple threads.
embOS for GNU is prepared to support the thread-local storage, but does not use it per default. This has the advantage of no additional overhead as long as thread-local storage is not needed by the application or specific tasks.
The embOS implementation of thread-local storage allows activation of TLS separately for every task.
Only tasks that call functions using TLS need to activate the TLS by defining a local variable and calling an initialization function when the task is started.
The _reent structure is stored on the task stack and have to be considered when the task stack size is defined. The structure may contain up to 800 bytes.

Typical Library objects that need thread-local storage when used in multiple tasks are:

- error functions -- errno, strerror.
- locale functions -- localeconv, setlocale.
- time functions -- asctime, localtime, gmtime, mktime.
- multibyte functions -- mbrlen, mbrtowc, mbsrtowc, mbtowc, wcrtomb, wcsrtomb, wctomb.
- rand functions -- rand, srand.
- etc functions -- atexit, strtok.
- C++ exception engine.

4.2.1 OS_ExtendTaskContext_TLS()

Description

`OS_ExtendTaskContext_TLS()` may be called from a task which needs thread local storage to initialize and use Thread-local storage.

Prototype

```
void  OS_ExtendTaskContext_TLS(struct _reent * pReentStruct)
```

Parameter

pReentStruct is a pointer to the thread local storage. It is the address of the variable of type struct `_reent` which holds the thread local data.

Return value

None.

Additional Information

`OS_ExtendTaskContext_TLS()` shall be the first function called from a task when TLS should be used in the specific task. The function must not be called multiple times from one task. The thread-local storage has to be defined as local variable in the task.
The structure stored on the task stack is really big and my take up to 800 bytes or more. This additional space has to be considered when the task stack sizes are defined.

Example

```
void Task(void) {
  struct _reent TaskReentStruct;

  OS_ExtendTaskContext_TLS(&TaskReentStruct);*/
  while (1) {
    ...  /* Task functionality */
  }
}
```

Please ensure sufficient task stack to hold the `_reent` structure variable.

For details on the `_reent` structure, `_impure_ptr`, and library functions which require precautions on reentrance, refer to the GNU documentation.

4.2.2 OS_ExtendTaskContext_TLS_VFP()

Description

`OS_ExtendTaskContext_TLS_VFP()` has to be called as first function in a task, when thread-local storage and thread safe floatingpoint processor support is needed in the task.

Prototype

`void` OS_ExtendTaskContext_TLS_VFP(`struct _reent * pReentStruct)`

Parameter

pReentStruct is a pointer to the thread local storage. It is the address of the variable of type struct `_reent` which holds the thread local data.

Return value

None.

Additional Information

`OS_ExtendTaskContext_TLS_VFP()` shall be the first function called from a task when TLS and VFP should be used in the specific task.
The function must not be called multiple times from one task.
The thread-local storage should be defined as local variable in the task.
The task specific TLS management is generated as embOS task extension together with the storage needed for the VFP registers. The VFP registers are automatically saved onto the task stack when the task is suspended, and restored, when the task is resumed. Additional task extension by a call of `OS_ExtendTaskContext()` is impossible.

4.3 Heap definition with embOS

The standard memory layout most commonly used with GNU tool chains can not be used with embOS.
The standard GNU memory layout defines the end address of data as the start of the heap and the end of RAM as the top address of the system stack.
The default management function `_sbrk()` checks if there is enough margin between current stack and current heap address, by comparing the stack pointer value against the highest address of the heap.
This requires, that the stack pointer is always higher than the last address in the heap.
This does not work with embOS, because task stacks are located in the normal data area, or are allocated from the heap. When tasks are running, the stack pointer is always lower than the start of the heap, or is lower then the current heap pointer when more space from the heap is requested by the application.

With embOS, the heap has to be defined as separate section in the linker script file, and the `_sbrk()` function delivered with embOS has to be used.
`_sbrk()` is defined in `OS_Syscalls.c` which can be found in the Setup folder of the start project.
When dynamic memory allocation with heap shall be used in the application, `OS_Syscalls.c` has to be compiled and linked.

The heap section has to be defined in the linker file.
The heap area has to start with the linker generated variable `__heap_start__` (lower address) and has to end with `__heap_end__` (upper address). The location in memory does not care, it may be anywhere in RAM.

4.4 Reentrancy, thread safe heap management

The heap management functions in the system libraries are not thread-safe without implementation of additional locking functions.
The GCC library calls two hook functions to lock and unlock the mutual access of the heap-management functions.
The empty locking functions from the system library may be overwritten by the application to implement a locking mechanism.

A locking is required when multiple tasks access the heap, or when objects are created dynamically on the heap by multiple tasks.
The locking functions are implemented in the source module `OS_MallocLock.c` which is included in the "Setup" subfolder in every embOS start project.
If thread safe heap management is required, the module has to be compiled and linked with the application.

4.4.1 __malloc_lock(), lock the heap against mutual access

`__malloc_lock()` is the locking function which is called by the system library whenever the heap management has to be locked against mutual access.
The implementation delivered with embOS claims a resource semaphore.

4.4.2 __malloc_unlock()

`__malloc_unlock()` is the is the counterpart to `__malloc_lock()`.
It is called by the system library whenever the heap management locking can be released. The implementation delivered with embOS releases the resource semaphore.

None of these functions has to be called directly by the application. They are called from the system library functions when required.
The functions are delivered in source form to allow replacement of the dummy functions in the system library.

4.5 Vector floating point support VFP

Some ARM MCUs come with an integrated vectored floating point unit VFP.
When selecting the CPU and activating the VFP support in the project options, the compiler and linker will add efficient code which uses the VFP when floating point operations are used in the application.
With embOS, the VFP registers have to be saved and restored when preemptive or cooperative task switches are performed.
For efficiency reasons, embOS does not save and restore the VFP registers for every task automatically. The context switching time and stack load are therefore not affected when the VFP unit is not used or needed.
Saving and restoring the VFP registers can be enabled for every task individually by extending the task context of the tasks which need and use the VFP.

4.5.1 OS_ExtendTaskContext_VFP()

Description

`OS_ExtendTaskContext_VFP()` has to be called as first function in a task, when the VFP is used in the task and the VFP registers have to be added to the task context.

Prototype

```
void OS_ExtendTaskContext_VFP(void)
```

Return value

None.

Additional Information

`OS_ExtendTaskContext_VFP()` extends the task context to save and restore the VFP registers during context switches.
Additional task context extension for a task by calling `OS_ExtendTaskContext()` is not allowed and will call the embOS error handler `OS_Error()` in debug builds of embOS.
There is no need to extend the task context for every task. Only those tasks using the VFP for calculation have to be extended.
When Thread-local Storage (TLS) is also needed in a task, the new embOS function `OS_ExtendTaskContext_TLS_VFP()` has to be called to extend the task context for TLS and VFP.

4.5.2 Using the VFP in interrupt service routines

Using the VFP in interrupt service routines requires additional functions to save and restore the VFP registers.
As the GCC compiler does not add additional code to save and restore the VFP registers on entry and exit of interrupt service routines, it is the users responsibility to save the VFP registers on entry of an interrupt service routine when the VFP is used in the ISR.
embOS delivers two functions to save and restore the VFP context in an interrupt service routine.

4.5.2.1 OS_VFP_Save()

Description

`OS_VFP_Save()` has to be called as first function in an interrupt service routine, when the VFP is used in the interrupt service routine. The function saves the temporary VFP registers on the stack.

Prototype

void OS_VFP_Save(void)

Return value

None.

Additional Information

`OS_VFP_Save()` declares a local variable which reserves space for all temporary floating point registers and stores the registers in the variable.
After calling the `OS_VFP_Save()` function, the interrupt service routine may use the VFP for calculation without destroying the saved content of the VFP registers.
To restore the registers, the ISR has to call `OS_VFP_Restore()` at the end.
The function may be used in any ISR regardless the priority.

4.5.2.2 OS_VFP_Restore()

Description

`OS_VFP_Restore()` has to be called as last function in an interrupt service routine, when the VFP registers were saved by a call of `OS_VFP_Save()` at the beginning of the ISR. The function restores the temporary VFP registers from the stack.

Prototype

```
void OS_VFP_Restore(void)
```

Return value

None.

Additional Information

`OS_VFP_Restore()` restores the temporary VFP registers which were saved by a previous call of `OS_VFP_Save()`.
It has to be used together with `OS_VFP_Save()` and should be the last function called in the ISR.

Example of an interrupt service routine using VFP:

```
void ADC_ISR_Handler(void) {
  OS_VFP_Save();          // Save VFP registers
  DoSomeFloatOperation();
  OS_VFP_Restore();       // Restore VFP registers
}
```

4.6 Cortex A VFP and Neon support

Some Cortex A ARM MCUs come with VFP/Neon unit.
When selecting the CPU and activating the VFP/Neon support, the compiler and linker will add efficient code which uses the VFP/Neon unit when floating point operations are used in the application.
With embOS, the VFP/Neon registers have to be saved and restored when preemptive task switches are performed.
For efficiency reasons, embOS does not save and restore the VFP/Neon registers for every task automatically. The context switching time and stack load are not affected. Saving and restoring the VFP/Neon registers can be enabled for every task individually be extending the task context of the tasks, where VFP/Neon unit is used.

4.6.1 OS_ExtendTaskContext_NEON()

Description

`OS_ExtendTaskContext_NEON()` has to be called as first function in a task, when the VFP/Neon unit is used in the task and the VFP/Neon regsisters have to be added to the task context.

Prototype

```
void OS_ExtendTaskContext_NEON(void)
```

Return value

None.

Additional Information

`OS_ExtendTaskContext_NEON()` extends the task context to save and restore the VFP/Neon registers for Cortex A CPUs during context switches.
Additional task context extension for a task by calling `OS_ExtendTaskContext()` is not allowed and will call the embOS error handler `OS_Error()` in debug builds of embOS.
There is no need to extend the task context for every task. Only those tasks using the VFP/Neon unit for calculation have to be extended.

4.6.2 Using the VFP/NEON in interrupt service routines

Using the VFP/Neon in interrupt service routines requires additional functions to save and restore the VFP/Neon registers.
As the GCC compiler does not add additional code to save and restore the VFP/Neon registers on entry and exit of interrupt service routines, it is the users responsibility to save the VFP/Neon registers on entry of an interrupt service routine when the VFP/Neon is used in the ISR.
embOS delivers two functions to save and restore the VFP/Neon context in an interrupt service routine.

4.6.2.1 OS_NEON_Save()

Description

`OS_NEON_Save()` has to be called as first function in an interrupt service routine, when the VFP/Neon is used in the interrupt service routine. The function saves the temporary VFP/Neon registers on the stack.

Prototype

```
void OS_NEON_Save(void)
```

Return value

None.

Additional Information

`OS_NEON_Save()` declares a local variable which reserves space for all temporary floating point registers and stores the registers in the variable.
After calling the `OS_NEON_Save()` function, the interrupt service routine may use the VFP/Neon for calculation without destroying the saved content of the VFP/Neon registers.
To restore the registers, the ISR has to call OS_NEON_Restore() at the end.
The function may be used in any ISR regardless the priority.

4.6.2.2 OS_NEON_Restore()

Description

`OS_NEON_Restore()` has to be called as last function in an interrupt service routine, when the VFP/Neon registers were saved by a call of `OS_NEON_Save()` at the beginning of the ISR. The function restores the temporary VFP/Neon registers from the stack.

Prototype

```
void OS_NEON_Restore(void)
```

Return value

None.

Additional Information

`OS_NEON_Restore()` restores the temporary VFP/Neon registers which were saved by a previous call of `OS_NEON_Save()`.
It has to be used together with `OS_NEON_Save()` and should be the last function called in the ISR.

Example of an interrupt service routine using VFP/NEON:

```
void ADC_ISR_Handler(void) {
  OS_NEON_Save();           // Save VFP/NEON registers
  DoSomeNeonOperation();
  OS_NEON_Restore();        // Restore VFP/NEON registers
}
```

4.6.3 Compiler and linker options.

The selection of different CPU cores or options like VFP support has to be done by linker, compiler and assembler options.
The options have to be passed to the tool by definitions in the make-file.
The options passed to the tools have to be defined for compiler, linker and assembler separately and have to be the same for all tools.
Beside other options, the most important options are the options to select the CPU core.

4.6.3.1 Options to select the Cortex-A9 core

```
-mfloat-abi=softfp "-mfpu=vfp" -march=armv7-a -mtune=cortex-a9 -mcpu=cortex-a9
```

Chapter 5

Stacks

5.1 Task stack for ARM

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 ARM 7/9, this minimum basic task stack size is about 68 bytes.

5.2 System stack for ARM

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.
For embOS, it is required, that the SVC stack is located in a specified section of specified size. The linker script has to define the symbol `__stack_svc_start__` at the beginning of thes stack (lower address) and `__stack_svc_end__` at the end of the IRQ stack (upper address).

5.3 Interrupt stack for ARM

If a normal hardware exception occurs, the ARM core switches to IRQ mode,which uses a separate stack pointer. To enable support for nested interrupts, saving regsiters onto the IRQ stack is necessary. embOS camoes with its own interrupt handler that 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. For the 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 isMaximum interrupt nesting level * 28 bytes.lp 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.
For embOS stack checking and for the scheduler, it is required, that the IRQ stack is located in a specified section of specified size.
The linker script has to define the symbol `__stack_irq_start__` at the beginning of thes stack (lower address) and `__stack_irq_end__` at the end of the IRQ stack (upper address).

5.4 Stack specifics of the ARM 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 modified to initialize the stacks.

Chapter 6

Interrupts

ARM7/ARM9/Cortex-A cores jump to a fixed location as a result of an exception or interrupt. Some microcontrollers may have an integrated vectored interrupt controller.
The following chapter describes how interrupts are handled by the core and how interrupt handler functions have to be defined in embOS.

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- As soon as interrupts are enabled, the interrupt is accepted.
- The CPU switches to the interrupt stack.
- The CPU saves the PC and flags into the registers `LR_irq` and `SPSR_irq`.
- The CPU jumps to the vector address `0x18` (or to offset 0x18 in the vector table) and continues execution from there.
- 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 `OS_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 ARM derivate with vectored interrupt controller, ensure that `IRQ_Handler()` is called from every interrupt. This is automatically done, when using an embOS start project which comes with embOS. The interrupt vector itself will then be examined by the C-level interrupt handler in `RTOSInit*.c`.

6.2 Defining interrupt handlers in C

Interrupt handlers called from the 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 `IRQ_Handler()`.

Depending on the interrupting source, it may be required to reset the interrupt pending condition of the related peripherals.

Example

Simple interrupt routine:

```
void Timer_irq_func(void) {
  if (__INTPND & 0x0800) {   // Interrupt pending ?
    __INTPND = 0x0800;       // reset pending condition
    OSTEST_X_ISR0();         // handle interrupt
  }
}
```

6.3 Interrupt handling with vectored interrupt controller

For ARM derivates with built in vectored interrupt controller, embOS uses a special interrupt handling procedure and delivers additional functions to install and setup interrupt handler functions.

When using an ARM derivate with vectored interrupt controller, ensure that `IRQ_Handler()` is called from every interrupt. This is default when the 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 interrupt handler `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 or vector number from the interrupt controller.
- `OS_irq_handler()` informs embOS that interrupt code is running by a call of `OS_EnterInterrupt() which does not re-enable interrupts, or a call of OS_EnterNestableInterrupt()` which re-enables interrupts.
- `OS_irq_handler()` calls the interrupt handler function which is addressed by the interrupt vector or interrupt ID and a vector table.
- `OS_irq_handler()` resets the interrupt controller to re-enable acceptance of new interrupts.
- `OS_irq_handler()` calls `OS_LeaveInterrupt()`/ `OS_LeaveNestableInterrupt()` which disables interrupts and informs embOS that interrupt handling has finished.
- `OS_irq_handler()` returns to IRQ_Handler().
- `IRQ_Handler()` restores registers and performs a return from interrupt.

Note: Different ARM CPUs may have different versions of vectored interrupt controller hardware, and usage of embOS supplied functions varies depending on the type of interrupt controller. Refer to the samples delivered with embOS which are used in the CPU specific RTOSInit module.
The ARM Generic interrupt controller (GIC) is supported by embOS and the embOS library deivers functions to access and control the GIC. These functions should not be called by the application directly, the embOS functions for handling vectored interrupt controller should be used.

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

Function	Description
`OS_ARM_InstallISRHandler()`	Installs an interrupt handler
`OS_ARM_EnableISR()`	Enables a specific interrupt
`OS_ARM_DisableISR()`	Disables a specific interrupt
`OS_ARM_ISRSetPrio()`	Sets the priority of a specific interrupt
`OS_ARM_AssignISRSource()`	Assigns a hardware interrupt channel to an interrupt vector. Available if required for specific CPU.
`OS_ARM_EnableISRSource()`	Enables an interrupt channel of a VIC type interrupt controller. Available if required for specific CPU.
`OS_ARM_DisableISRSource()`	Disables an interrupt channel of a VIC type interrupt controller. Available if required for specific CPU.

Table 6.1: Interrupt handler functions for ARM derivates with built in vectored interrupt controller

6.3.1 OS_ARM_InstallISRHandler(): Install an interrupt handler

Description

`OS_ARM_InstallISRHandler()` is used to install a specific interrupt vector when ARM 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 6.2: OS_ARM_InstallSRHandler() parameter list

Return value

`OS_ISR_HANDLER *`: The address of the previously 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.
If the interrupt controller delivers IDs only, the interrupt vector is written into a vector table in RAM. The vector table is a static variable in the RTOSInit module.

6.3.2 OS_ARM_EnableISR(): Enable a 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 6.3: 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.

Note: For ARM 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 call `OS_ARM_EnableISRSource()` also.
For ARM CPUs with GIC, this function enables the interrupt source in the GIC and assigns the running CPU core to the requested interrupt source in the interrupt distributor when multicore CPUs are used. It does not enable any perpheral.

6.3.3 OS_ARM_DisableISR(): Disable a 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 6.4: 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.

Note: When using an ARM CPU with built in interrupt controller of VIC type, use `OS_ARM_DisableISRSource()` to disable a specific interrupt.

For ARM CPUs with GIC, this function disables the interrupt source in the GIC only. It does not modifiy the interrupt distributor and does not disable any perpheral.

6.3.4 OS_ARM_ISRSetPrio(): Set priority of a 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 6.5: 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. Refer to CPU-specific manuals about allowed priority levels.

6.3.5 OS_ARM_AssignISRSource(): Assign a hardware interrupt channel to an interrupt vector

Description

`OS_ARM_AssignISRSource()` is used to assign a hardware interrupt channel to an interrupt vector in an interrupt controller of VIC type.

Prototype

```
void OS_ARM_AssignISRSource ( int ISRIndex,
                              int Source );
```

Parameter	Description
`ISRIndex`	Index of the interrupt source which should be modified.
`Source`	The source channel number which should be assigned to the specified interrupt vector.

Table 6.6: OS_ARM_AssignISRSource() parameter list

Additional Information

This function assigns a hardware interrupt line to an interrupt vector of VIC type only. It cannot be used for other types of vectored interrupt controllers. The hardware interrupt channel number of specific peripherals depends on specific CPU derivates and has to be taken from the hardware manual of the CPU.

This function is available for CPUs with the VIC type of vectored interrupt controller, it is not available on CPUs with other types of vectored interrupt controller.

6.3.6 OS_ARM_EnableISRSource(): Enable an interrupt channel of a VIC-type interrupt controller

Description

`OS_ARM_EnableISRSource()` is used to enable an interrupt input channel of an interrupt controller of VIC type.

Prototype

```
void  OS_ARM_EnableISRSource ( int SourceIndex );;
```

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

Table 6.7: OS_ARM_EnableISRSource() parameter list

Additional Information

This function enables a hardware interrupt input of a VIC-type interrupt controller. It cannot be used for other types of vectored interrupt controllers. The hardware interrupt channel number of specific peripherals depends on specific CPU derivates and has to be taken from the hardware manual of the CPU.

This function is available for CPUs with the VIC type of vectored interrupt controller, it is not available on CPUs with other types of vectored interrupt controller.

6.3.7 OS_ARM_DisableISRSource(): Disable an interrupt channel of a VIC-type interrupt controller

Description

`OS_ARM_DisableISRSource()` is used to disable an interrupt input channel of an interrupt controller of VIC type.

Prototype

```
void OS_ARM_DisableISRSource ( int SourceIndex );;
```

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

Table 6.8: OS_ARM_DisableISRSource() parameter list

Additional Information

This function disables a hardware interrupt input of a VIC-type interrupt controller. It cannot be used for other types of vectored interrupt controllers. The hardware interrupt channel number of specific peripherals depends on specific CPU derivates and has to be taken from the hardware manual of the CPU.

This function is available for CPUs with the VIC type of vectored interrupt controller, it is not available on CPUs with other types of vectored interrupt controller.

Example

```
/* Install UART interrupt handler */
OS_ARM_InstallISRHandler(UART_ID, &COM_ISR);     // UART interrupt vector
OS_ARM_ISRSetPrio(UART_ID, UART_PRIO);           // UART interrupt priotity
OS_ARM_EnableISR(UART_ID);                        // Enable UART interrupt

/* Install UART interrupt handler with VIC type interrupt controller*/
OS_ARM_InstallISRHandler(UART_INT_INDEX, &COM_ISR);    // UART interrupt vector
OS_ARM_AssignISRSource(UART_INT_INDEX, UART_INT_SOURCE);
OS_ARM_EnableISR(UART_INT_INDEX);               // Enable UART interrupt vector
OS_ARM_EnableISRSource(UART_INT_SOURCE);         // Enable UART interrupt source
```

6.4 Interrupt-stack switching

Because ARM 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 ARM interrupt stack is used for the primary interrupt handler in `RTOSVect.asm` only.

6.5 Fast Interrupt (FIQ)

The FIQ interrupt cannot be used with embOS functions, it is reserved for high speed user functions.

Note the following:

- FIQ is never disabled by embOS.
- Never call any embOS function from an FIQ handler.
- Do not assign any embOS interrupt handler to FIQ.

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

Chapter 7

STOP / WAIT Mode

7.1 Introduction

In case your controller supports some kind of power saving mode, it is 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 the function `OS_Idle()`, which you can find in embOS module `RTOSInit.c`.
Per default, in non debug builds, the CPU clock should be switched off in OS_Idle().

Chapter 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. The minimum ROM requirement for the kernel itself is about 1.700 bytes.

In the table below, which is for release build, you can find minimum RAM size requirements for embOS resources. Note that the sizes depend on selected embOS library mode.

embOS resource	**RAM [bytes]**
Task control block	52
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Software timer	20

Chapter 9

Files shipped with embOS

List of 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 the generic documentation.
Start\BoardSupport\		Sample project files for Eclips DS-5 workbench, contained in manufacturer specific sub folders.
Start\Inc	RTOS.h BSP.h	Include file for embOS, to be included in every C-file using embOS functions.
Start\Lib	libos*.a	embOS libraries for usage with GNU compiler.
Start\BoardSupport\..\Setup	OS_Error.c	embOS runtime error handler used in stack check or debug builds.
Start\BoardSupport\...\Setup\	*.*	CPU specific hardware routines, setup files, linker files, debug support files.

Any additional files shipped serve as example.

Index