embOS

Real-Time Operating System

CPU & Compiler specifics for Altera NIOS II CPUs using NIOS II Software Build Tools

Document: UM01035
Software Version: 5.8.2.0
Revision: 0
Date: January 24, 2020

A product of SEGGER Microcontroller GmbH

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 (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER 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

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: support@segger.com
Internet: www.segger.com

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

Print date: January 24, 2020

Software	Revision	Date	By	Description
5.8.2.0	0	200124	MC	Update to latest embOS generic sources.
4.12a	0	150917	MC	Update to latest embOS generic sources.
4.12a	0	150212	MC	Initial version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

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 program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Using embOS (NIOS II Software Build Tools)

Since version 9.1, Altera delivers the NIOS II Software Build Tools for Eclipse, which are installed as the default development platform for NIOS II. The older NIOS II IDE was still delivered with following versions of the software and can be used to develop an embOS application.

Assuming that you are using the NIOS II Software Build Tools to develop your application, follow the instructions in this chapter. If you decide to use the NIOS II IDE, follow the instructions in *Using embOS (NIOS II IDE)* on page 15 instead.

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.

1.2 First steps

After installation of embOS you can create your first multitasking application. Your embOS distribution contains the following folders needed for a project running under the NIOS II Software Build Tools:

- `Hello_Project`: contains all sources required for an embOS application.
- `Hello_BSP`: contains the embOS libraries and additional files required to add embOS support into the BSP and application.

1.3 Setup the development environment

- Start the NIOS II Software Build Tools for Eclipse.
- Create a new Hello World project plus BSP from template.
- Close the NIOS II Software Build Tools.
- Copy everything from the `Hello_Project\` folder into the `hello_world_x\` folder that was created by the NIOS II Software Build Tools:
  - Copy the entire `CPU\` folder from the `Hello_Project\` folder delivered with embOS into the `hello_world_x\` folder which was created by by the NIOS II Software Build Tools.
  - Copy the entire Src folder from the `Hello_Project\` folder delivered with embOS into the `hello_world_x\` folder which was created by the NIOS II Software Build Tools.
- Copy everything from the `Hello_BSP\` folder into the `hello_world_bsp\` folder that was created by the NIOS II Software Build Tools:
  - Copy the entire `drivers\` folder from the `Hello_BSP\` folder delivered with embOS into the `hello_world_bsp\` folder which was created by the NIOS II Software Build Tools.
  - Copy the entire `HAL\` folder from the `Hello_BSP` folder delivered with embOS into the `hello_world_bsp\` folder which was created by the NIOS II Software Build Tools.
- Finally delete the `hello_world.c` file in the `hello_world_x` folder that was created by the NIOS II Software Build Tools.

1.4 Prepare and build the sample start application

- Start the NIOS II Software Build Tools for Eclipse again.
- Select and refresh the BSP. This should include all new sources that were copied into the `hello_world_bsp\` folder.
- Select and refresh the project. This should include all new sources that were copied into the `hello_world project\` folder.
- Modify the Makefile of your `hello_world_x\` project:
  - Edit the the search path for libraries, `ALT_LIBRARY_DIRS`:
  The embOS libraries are located in the drivers folder that was copied into the bsp folder of the project. Assuming the bsp folder was named `hello_world_bsp\` during project creation, and the bsp folder is on the same level as the project folder in the directory structure, the search path to the libraries is relative to the project folder and can be set as relative path as follows:
  `ALT_LIBRARY_DIRS := ../hello_world_bsp/drivers/segger_embOS/embosLibs`
  - Add the library names for the `-msys-lib` linker option to add an embOS library as system library:
  `ALT_BSP_DEP_LIBRARY_NAMES := embOS_DP`

to add the embOS debug and profiling library to the project.
- Verify the library selection defined in `OS_Config.h`:
`#define OS_LIBMODE_DP`

- Save the Makefile.
- Rebuild the project.

You may now start the NIOS II Hardware debugger to download the application into your CPU and step through the application. Initially, a target for debug and profiling is built. It includes debug information to be used with the NIOS II debugger and may be run on a NIOS II using the JTAG interface, ByteBlaster or USB-Blaster.

1.5 Build your own application

To build your own application, you may start with the sample hello world project that was built as start. This has the advantage that all necessary files are included and all settings for the project are already done. You may alternatively start with a project and BSP that you already have for your application and add the embOS files as described in *First steps* on page 10.

1.6 Required files for an embOS application

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

- `RTOS.h` from subfolder `HAL\inc\` that was copied into the project specific BSP folder. This header declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- `RTOSInit.c` from subfolder `CPU\`. It contains hardware dependent initialization code for the embOS timer and optional UART for embOSView.
- `OS_Error.c` from the subfolder `Src\`. The embOS error handler is defined in this file. It is required when stack check or runtime error check for debug libraries is required.
- One library from the subfolder `drivers\segger_embos\embosLibs\` in the BSP.
- All other files from the `Hello_BSP` folder, which have to be copied into the bsp folder of your project.

1.7 Add your own code

You may add your own code in any subfolder of the hello world project folder. After refreshing your project, your code should be included.

1.8 Change the embOS library mode

For your application you may wish to choose an other embOS library type. During development, you should use an embOS debug library. For your final application you may wish to use an embOS release library. Select the library by modification of the `ALT_BSP_DEP_LIBRARY_NAMES` definition in the Makefile of the application. This switch is used to set the `-msys-lib` linker option and has to select an embOS library as system library.

One of the following settings for `ALT_BSP_DEP_LIBRARY_NAMES` can be used:

1. `embOS_XR` to select the eXtreme Release library `libembOS_XR.a`
2. `embOS_R` to select the Release library `libembOS_R.a`
3. `embOS_S` to select the Stack check library `libembOS_S.a`
4. `embOS_SP` to select the Stack check and Profiling library `libembOS_SP.a`
5. `embOS_D` to select the Debug library `libembOS_D.a`
6. `embOS_DP` to select the Debug Profiling library `libembOS_DP.a`
7. `embOS_DT` to select the Debug Trace library `libembOS_DT.a`

The embOS library mode has to be defined as project option according to the embOS library which was chosen as system library. This is required to select library dependent functionality and data types in the `RTOS.h` file.

The library mode has to be set for the BSP and the project. The library mode should be defined in the `OS_Config.h` file which is included in the embOS API header file `RTOS.h`.

1.9 Re-building or modifying the BSP using NIOS II

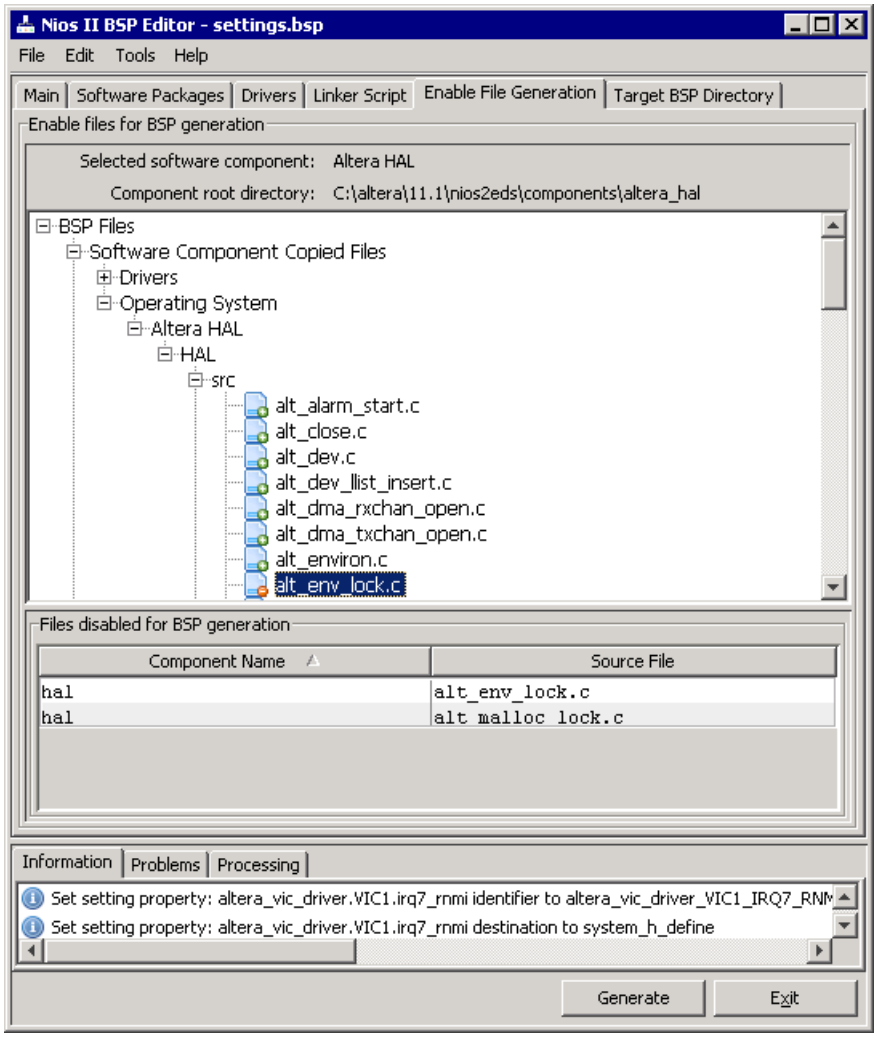
BSP Editor The BSP Editor is used to set or adjust major project options such as linker settings, resource usage, system timer an others. After creation of the BSP and adding embOS support by copying the files from the `embOS_BSP\` folder into the BSP folder of the project, it may be required to call the BSP Editor to adjust some settings.

> **Note**
>
> Whenever the NIOS II BSP Editor is closed with modified settings, the BSP is regenerated. Some sources are created and some generic sources are copied from an Altera template folder. These generic sources replace the BSP specific sources required for embOS and embOS will not be included in the project.

There are two possible solutions to fix or avoid this problem:

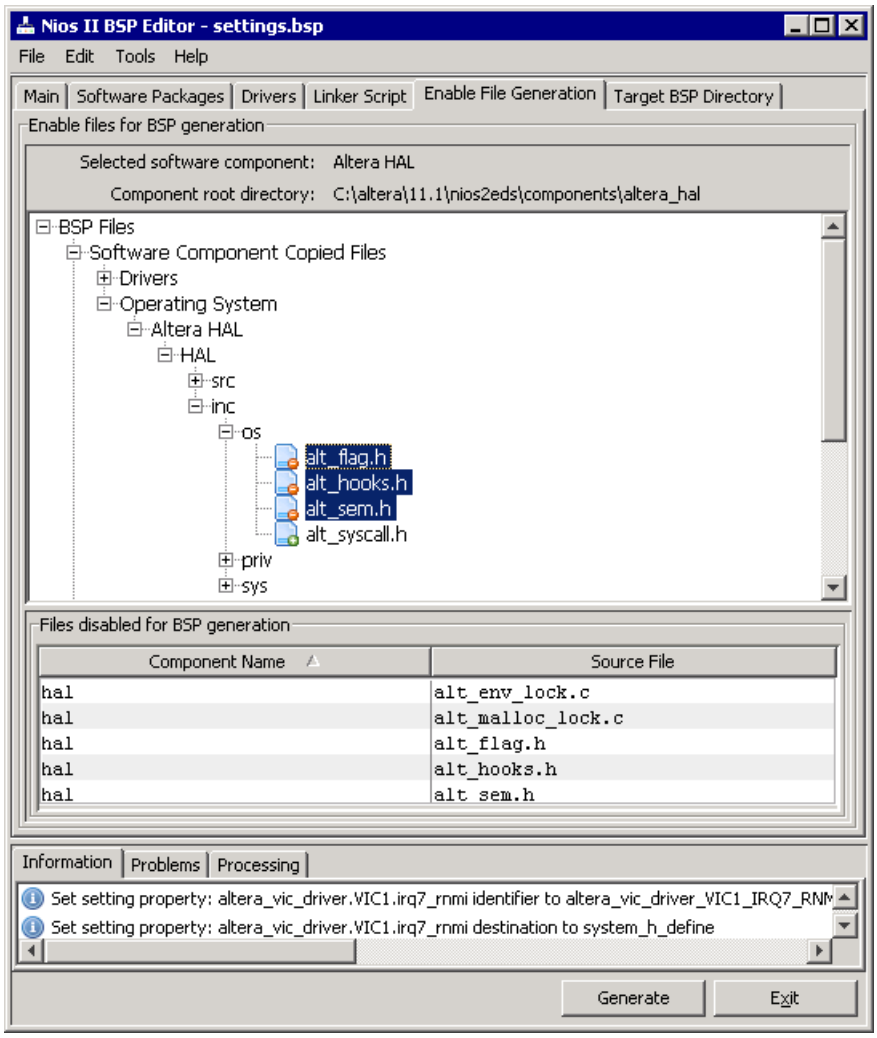
Solution 1

To avoid problems with the BSP Editor, you may disable the file generation for the specific BSP files which were delivered with embOS. Open the Nios II BSP Editor. Select the tab `Enable File Generation` and disable two source files for BSP generation. Browse through `Software Component Copied Files -> Operating Sytem -> HAL -> src`, select the files `alt_malloc_lock.c` and `alt_env_lock.c` and then, after right click, select `Disable generation` from the context menu.



After disabling the two source files from generation, three additional header files which were delivered with embOS have to be disabled from generation. Browse through `Software Component Copied Files -> Operating Sytem -> HAL -> inc -> os`, select the

files `alt_flag.h`, `alt_hooks.h` and `alt_sem.h` and then, after right click, select Disable generation from the context menu.



Solution 2

The second method has to be used when BSP was regenerated without disabling the embOS specific files as described above. After editing the BSP or after regeneration of the BSP, copy all files from the `Hello_BSP\` folder delivered with embOS into the BSP folder of your project.

- Copy the entire `drivers\` folder from the `Hello_BSP\` folder delivered with embOS into the `BSP\` folder of your project.
- Copy the entire `HAL\` folder from the `Hello_BSP\` folder delivered with embOS into the `BSP\` folder of your project.

Chapter 2

Using embOS (NIOS II IDE)

Since version 9.1, Altera delivers the NIOS II Software Build Tools for Eclipse, which are installed as the default development platform for NIOS II. The older NIOS II IDE was still delivered with following versions of the software and can be used to develop an embOS application.

2.1 Installation

embOS is shipped as a zip-file in electronic form. To install it, proceed as follows: Extract it to any folder of your choice, preserving the directory structure of the zip-file.

2.2 First steps

After installation of embOS you can create your first multitasking application. Your embOS distribution contains the following folders needed for a project running under the NIOS II IDE:

- `Hello_Project`: contains all sources required for an embOS application.
- `components`: contains the embOS libraries and additional components used to integrate embOS into the HAL environment for NIOS II CPUs.
- `Hello_BSP`: contains the embOS libraries and additional files required to add embOS support into the BSP and application.

2.2.1 Setup the development environment

- Start the NIOS II IDE.
- Create a new Hello World project.
- Close the NIOS II IDE.
- Copy everything from the `Hello_Project\` folder into the `hello_world_x\` folder that was created by the NIOS II IDE:
  - Copy the entire `CPU\` folder from the `Hello_Project\` folder delivered with embOS into the `hello_world_x\` folder that was created by by the NIOS II IDE.
  - Copy the entire `Src\` folder from the `Hello_Project\` folder delivered with embOS into the `hello_world_x\` folder that was created by the NIOS II IDE.
- Copy everything from the `components\` folder delivered with embOS into the `components\` folder of your `nios2eds\components\` folder. Usually, this is located at `altera\91\nios2ed\` after a standard installation of the NIOS II software tools 9.1
- Copy everything from the `Hello_BSP\` folder into the `hello_world_x_syslib\` folder that was created by the NIOS II IDE:
  - Copy the "C" source files from the `Hello_syslib\` folder delivered with embOS into the `hello_world_x_syslib\` folder that was created by the NIOS II IDE.
  - Copy the entire `os\` folder from the `Hello_syslib\` folder delivered with embOS into the `hello_world_x_syslib\` folder that was created by the NIOS II IDE.
  - Copy the entire `priv\` folder from the `Hello_syslib` folder delivered with embOS into the `hello_world_x_syslib\` folder that was created by the NIOS II IDE.
- Finally delete the `hello_world.c` file in the `hello_world_x\` folder that was created by the NIOS II IDE.

2.2.2 Prepare and build the sample start application

- Start the NIOS II IDE again.
- Modify the project settings for your `hello_world_x` project:
  - Under `Properties | C/C++ Build | NIOS II Compiler | General | Include Paths` add the embOS include path for `RTOS.h`:
  `C:\altera\91\nios2eds\components\segger_embos\embOS\inc`
  You may alternatively browse for the path.
  - Under `Properties | C/C++ Build | NIOS II Compiler | Preprocessor | Defined Symbols` add the symbol for embOS library mode:
  `OS_LIBMODE_DP`.
  - Under `Properties | C/C++ Build | Linker | General | Linker Flags` add the flag to link the embOS system library:
  `-msys-lib = embOS_DP`
  - Under `Properties | C/C++ Build | Linker | General | Library Paths` add the path to the embOS library:
  `C:\altera\91\nios2eds\components\segger_embos\embOS\embOSLibs`
  You may alternatively browse for the path.

- Modify the related hello_world_x-syslib properties:
  - Under C/C++ Build | NIOS II Compiler | General | Include Paths add the path to the embOS API header file RTOS.h:
  C:\altera\91\nios2eds\components\segger_embos\embOS\inc
  You may alternatively browse for the path.
  - Under Properties | C/C++ Build | NIOS II Compiler | Preprocessor | Defined Symbols add the symbol for the embOS library mode:
  OS_LIBMODE_DP
- Refresh your project (press F5). This should include all new sources which are contained in the Src\ and CPU\ folders which were copied into your project folder.
- Rebuild your project.

You may now step through your project using the NIOS II simulator debugger or hardware debugger. Initially a target for debug and profiling is built. It includes debug information to be used with NIOS debugger. It may be run on NIOS2 using JTAG interface.

2.3 Build your own application

To build your own application, you may start with the sample hello world project that was built as start. This has the advantage that all necessary files are included and all settings for the project are already done.

2.4 Required files for an embOS application

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

- RTOS.h from subfolder components\segger_embos\embOS\inc\. This header declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- RTOSInit.c from subfolder CPU\. It contains hardware dependent initialization code for the embOS timer and optional UART for embOSView.
- OS_Error.c from the subfolder Src\. The embOS error handler is defined in this file. It is required when stack check or runtime error check for debug libraries is required.
- One library from the subfolder components\segger_embos\embOS\embosLibs\.
- All files from the Hello_syslib\ folder, which have to be copied into the syslib\ folder of your project.

2.5 Add your own code

You may add your own code in any subfolder of the hello world project folder. After refreshing your project, your code should be included.

2.6 Change the embOS library mode

For your application you may wish to choose an other embOS library type. During development, you should use an embOS debug library. For your final application you may wish to use an embOS release library. Therefore you have to replace the embOS library define macro in your project and have to modify the linker flag -msys-lib to address the correct library.

Chapter 3

Sample Application

3.1 The sample application OS_StartLEDBlink.c

The following is a printout of the sample application `OS_StartLEDBlink.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. `OS_InitHW()` is in the `RTOSInit*.c` file that is delivered as source and initializes the CPU if required. The embOS system itself was automatically initialized by the Altera HAL during startup before main was called. After creation of the tasks, `OS_Start()` is called to start the kernel. `OS_Start()` never returns. It activates the task with the highest priority and starts it. In our example, two tasks are activated and execute until they run into a delay, then suspend for the specified time and continue execution afterwards.

```
#include "RTOS.h"
#include "BSP.h"

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

static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_TASK_Delay(50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_TASK_Delay(200);
  }
}

/*************************************************************************
*
*       main()
*/
int main(void) {
  OS_Init();    // Initialize embOS
  OS_InitHW();  // Initialize required hardware
  BSP_Init();   // Initialize LED ports
  OS_TASK_CREATE(&TCBHP, "HP Task", 100, HPTask, StackHP);
  OS_TASK_CREATE(&TCBLP, "LP Task",  50, LPTask, StackLP);
  OS_Start();   // Start embOS
  return 0;
}
```


Chapter 4

Libraries

This chapter includes CPU-specific information such as CPU-modes and available libraries.

4.1 Data / Memory models, compiler options

embOS for NIOS II CPUs is delivered with different libraries and can be used for all different NIOS II CPU options.

4.2 Available library modes

There are seven different libraries available.

Features	Define
Extreme Release	`OS_LIBMODE_XR`
Release	`OS_LIBMODE_R`
Stack check	`OS_LIBMODE_S`
Stack check + profiling	`OS_LIBMODE_SP`
Debug + stack check	`OS_LIBMODE_D`
Debug + stack check + profiling	`OS_LIBMODE_DP`
Debug + stack check + profiling + trace	`OS_LIBMODE_DT`

The appropriate define has to be passed to the compiler. This can be done with a preprocessor option in your project and syslib settings for the NIOS II IDE, or by an appropriate define in the `OS_Config.h` file when using the NIOS II Software Build Tools.

If no library mode is defined, `OS_Config.h` is included which allows a default library mode definition in this file.

Chapter 5

CPU and compiler specifics

The hardware initialization routines and default settings in `RTOSInit.c` were designed for NIOS II CPUs using the sample standard setup delivered with Altera NIOS II Development Kits.

All CPU hardware dependent routines are found in `RTOSInit.c` and in the additional header and source files used for the HAL setup in your components subdirectory. To use embOS with the NIOS II IDE, please ensure the embOS components are copied to your components directory and the specific files for your syslib are copied into your current syslib folder as described under *Installation* on page 16. As you may have built your own CPU, hardware dependent routines in `RTOSInit.c` may have to be modified for your particular CPU.

5.1 Clock settings for embOS timer interrupt

`OS_InitHW()` routine in `RTOSInit.c` does not initialize an embOS timer tick, as this should have been done during initialization of the HAL environment. Normally a system timer tick rate of 1 msec should have benn used per default. You may modify this tickrate in `OS_InitHW()` if required. If you modified the system timer, or did not use the default system timer, you have to modify:

- `OS_InitHW()` to initialize a timer for usage with embOS.
- `OS_ISR_Tick()` which resets timer interrupt pending and handles embOS timer interrupts.

embOS will run without any timer, but you will loose any time related functions or software timer. If there is only one hardware timer available in your CPU and you need this for your own code, you may add your code into the embOS timer tick handler `OS_ISR_Tick()` in `RTOSInit.c`.

5.2 Settings for UART used for embOSView

`OS_COM_Init()` routine in `RTOSInit.c` was written for sample CPUs of NIOS II Development Kit, Stratix edition. This UART itself does not require any init routines, baudrate is fixed to 115200 baud and can not be changed.

The only thing what is done here is:

- Interrupt handler for UART interrupt is installed
- Interrupts for Rx and Tx are enabled.

If your particular UART differs, or if you may want to use an other UART for embOSView, you have to modify:

- `OS_COM_Init()` to initialize UART for reception and transmission.
- `OS_COM_Send1()` to send one byte via UART
- `OS_COM_Isr()` to handle Rx and Tx interrupts.

5.3 Reentrancy

Some GNU library functions used for NIOS II are not reentrant. To solve the reentrancy problem of those functions, an individual structure containing all relevant data is required for every task calling non reentrant functions. This data structure is defined as `struct _reent` in the NIOS II development environment. Because this structure contains several hundred bytes that have to be stored in the task stack, embOS does not automatically implement this reentrancy data for every task upon creation of a task.

When you call non-reentrant library functions from a task, you have to create a `_reent` structure on the task stack and have to initialize an associated pointer. Define a local reent structure variable as first variable in the task function and then initialize it by simply calling `OS_InitReent()`, which is delivered with embOS in the source file `OS_InitReent.c`.

Example

```
void Task(void) {
  struct _reent TaskReentStruct;  /* Only required if task calls     */
  OS_InitReent(&TaskReentStruct); /* non-reentrant library functions */
  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, please refer to the Altera documentation.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- The CPU context is saved.
- The Altera HAL low level interrupt function is called.
- The interrupt handler which was installed by a HAL interrupt installation function is called. This function dispatches to the assigned interrupt handler function.
- The interrupt handler handles the interrupt and returns.
- The Altera HAL low level imterrupt handler restores CPU context.
- The interrupt function continues operation.

For details, refer to the NIOS II CPU user manuals and appliaction notes which are delivered with your NIOS II tools.

6.2 Defining interrupt handlers in C

Interrupt handlers are written as normal functions. The interrupt handler model used by NIOS II GNU tools allows parameters passed to interrupt handlers. Using the HAL with embOS, the parameters are not passed to the high level interrupt handler.

Altera HAL implements an embOS-safe interrupt handler which automatically informs embOS that interrupt code is running. Therefore your interrupt handlers do not need to call `OS_INT_Enter()` / `OS_INT_Leave()` as it is required by other embOS ports and described in the generic manuals, but it may be used to clarify that an interrupt handler is called that does not re-enable interrupts.

Interrupts must not be re-enabled during execution of an interrupt handler, except for when the special embOS function `OS_INT_EnterNestable()` is used as first instruction in an interrupt handler.

Examples

Simple interrupt handler:

```
void ISR_Timer(void* context, alt_u32 id) {
  /* OS_INT_EnterNestable(); does not have to be called */
  _HandleTimer();
  /* OS_INT_LeaveNestable(); does not have to be called */
}
```

Interrupt handler which re-enables interrupts:

```
void ISR_Timer(void* context, alt_u32 id) {
  OS_INT_EnterNestable();  /* Inform embOS that interrupt handler */
                           /* is running and re-enable interrupts */
  _HandleTimer();
  OS_INT_LeaveNestable();
}
```

6.3 Interrupt-stack switching

Separate interrupt stack and interrupt stack switching for NIOS II is not implemented as embOS function. `OS_INT_EnterIntStack()` and `OS_INT_LeaveIntStack()` are supplied for source compatibility to other processors and future use. They have no functionality. The low level interrupt handler which is implemented in the Altera HAL allows usage of a separate interrupt stack which may be located in different RAM locations. By setting the option “Use a separate exception stack” in the System Library settings, the low level interrupt handler switches to the separate stack during exception execution. This option does not work with embOS as long as the vectored interrupt controller is not used and must not be used. An error should be generated when a project is built with this option.

6.4 Interrupt priorities

Interrupt priorities of peripherals are hardwired and can not be changed during runtime as long as the VIC is not used. embOS Timer and UART may run on lowest priority. User interrupt handler calling embOS functions may run on any priority.

6.5 Vectored Interrupt Controller VIC

Newer versions of the NIOS II Software Build Tools allow usage of a vectored interrupt controller in the CPU. The vectored interrupt controller has the advantage of faster response to peripheral interrupts. The VIC can be used with embOS version 3.84 or later.

There are no modifications required in the application. The code which supports the VIC is automatically inserted from the embOS library when the VIC is defined in the project.

6.5.1 Nested interrupts with the vectored interrupt controller

The vectored interrupt controller allows nested interrupts, when the macro `ALTERA_VIC_DRIVER_PREEMPTION_INTO_NEW_REGISTER_SET_ENABLED` is defined.

Nested interrupts with VIC may be used with embOS. The required code for nested interrupt support with VIC is included in the embOS libraries. When using the VIC with nested interrupts, the high level interrupt handler functions do not need to be modified.

Usage of the embOS functions `OS_INT_EnterNestable()` and `OS_INT_LeaveNestable()` is not required in the interrupt handler, the functions must not be called when using the VIC with nested interrupt support.

6.5.2 Quick nested interrupts with the vectored interrupt controller

High speed nested interrupts (quick nested interrupts) with the vectored interrupt controller can be activated by a call of:

```
NIOS2_WRITE_CONFIG(NIOS2_CONFIG_REG_ANI_MASK);
```

This option shall not be used with embOS because it may fail with the default interrupt handler from Altera.

Chapter 7

Stacks

This chapter describes how embOS uses the different stacks of the NIOS II CPU.

7.1 Task stack for NIOS II

Each task uses its individual stack. The stack pointer is initialized and set every time a task is activated by the scheduler. The stack-size required for a task is the sum of the stack-size of all routines, plus a basic stack size, plus size used by exceptions.

The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by calling embOS-routines.

For NIOS II CPUs, this minimum basic task stack size is about 52 bytes. Because any function call uses some amount of stack and every exception also pushes several bytes onto the current stack, the task stack size has to be large enough to handle at least one exception too. We recommend at least 512 bytes stack as a start, because a separate exception stack must not be activated when using embOS.

7.2 System stack for NIOS II

The system stack is the stack used during startup before the call of `OS_Start()`. The system stack is also used during embOS task switches, for embOS timer and internal functions.

Startup code normally initializes the stack pointer.

To enable stack check of embOS a size of the system stack has to be defined. This is done by a constant define `SYS_STACK_SIZE` which may be redefined or modified by compiling embOS sources.

Per default, `SYS_STACK_SIZE` is defined for 512 bytes of system stack.

7.3 Interrupt stack for NIOS II

NIOS II CPUs do not implement a separate interrupt stack by hardware. Every interrupt runs on the current stack, as long as stack switching to separate exception stack is not used. The stack used for interrupts and exceptions is either the task stack or the system stack.

Therefore task stacks have to be sufficient to handle all nested interrupts. Interrupt stack switching by embOS is not implemented.

The separate exception stack that may be enabled by system library project or BSP settings can not be used with embOS when the default interrupt controller and handler is used. An error is generated during build if the separate exception stack is enabled.

Chapter 8

Technical data

This chapter lists technical data of embOS used with Altera NIOS II CPUs.

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 3.000 bytes.

In the table below, which is for X-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	36
Software timer	20
Mutex	16
Semaphore	8
Mailbox	24
Queue	32
Task event	0
Event object	16

Chapter 9

Shipped Files

embOS for NIOS II CPUs and NIOS II Software Tools is shipped with documentation in PDF format and release notes as html.

All source files and additional files which are required for embOS used with the NIOS II IDE are located in the subfolders `Hello_Project\`, `Hello_syslib\` and components. All source files and additional files which are required for embOS used with the NIOS II Software Build Tools are located in the subfolders `Hello_Project\` and `Hello_BSP\`.

The source version of embOS comes with an additional folder `GenOSSrc\` , which contains the generic embOS sources, and a folder `CPU\` , which contains CPU specific files required to recompile the libraries.

embOSView, release notes and the manuals are found in the root directory of the distribution.

9.1 List of files shipped with embOS

File	Explanation
root	
*.pdf	Generic API and target specific documentation.
Release_embOS_NIOS2_GNU.html	Version control document.
embOSView.exe	Utility for runtime analysis, described in generic documentation.
components	
.	Source files, headers, libraries and additional files required for embOS used with the NIOS II IDE.
Hello_BSP	
.	Source files, headers and libraries required for embOS used with the NIOS II Software Build Tools.
Hello_Project\CPU	
RTOSInit.c	Source file required for embOS used with both the NIOS II IDE and the NIOS II Software Build Tools.
Hello_Project\Src	
OS_StartLEDBlink.c OS_Error.c OS_InitReent.c	Source files required for embOS used with both the NIOS II IDE and the NIOS II Software Build Tools.
Hello_syslib	
.	Source and header files required for embOS used with the NIOS II IDE.

Any additional files shipped serve as example.

